Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Resolução de Horários de Exames com Pesquisa Local Restringida

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Francisco José de Abreu Roldão

Orientador: Prof. Doutor Pedro Barahona

Júri:

Presidente: Prof. Doutor José Augusto Legatheaux Martins
Arguente: Prof. Doutora Paula Alexandra da Costa Amaral
Vogal: Prof. Doutor Pedro Manuel Corrêa Calvente de Barahona

Novembro de 2011

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Resolução de Horários de Exames com Pesquisa Local Restringida

Francisco José de Abreu Roldão

Orientador: Prof. Doutor Pedro Barahona

Trabalho apresentado no âmbito do Mestrado em Engenharia Informática, como requisito parcial para obtenção do grau de Mestre em Engenharia Informática.

Novembro de 2011

Resolução de Horários de Exames com Pesquisa Local Restringida

Resumo

A produção de horários é um problema altamente combinatório dado o conjunto de restrições que envolve e as suas interdependências, bem como um conjunto de preferências e características de qualidade que são difíceis de especificar e mais ainda de quantificar.
Não sendo razoável explorar todo o espaço de pesquisa dada a sua dimensão exponencial, a utilização de pesquisa local torna-se apelativa, em parte porque a definição de vizinhanças pode ser feita de uma forma bastante intuitiva, permitindo a validação e comparação com métodos manuais de resolução do problema. Recentemente, começam a aparecer linguagens e ferramentas para pesquisa local restringida que favorecem a declaratividade da especificação de pesquisa local e que se têm revelado altamente competitivas na resolução de vários tipos de problemas combinatórios.
Nesta dissertação, é testada esta abordagem na produção de horários. Mais especificamente, são analisadas o tipo de restrições, preferências e medidas de qualidade que ocorrem na especificação de horários. Subsequentemente, após a análise das vizinhanças a utilizar na resolução destes problemas, implementou-se um protótipo, bem como as meta-heurísticas que se revelaram mais interessantes.
O protótipo implementado foi testado com exemplos de horários produzidos na FCT/UNL e as várias meta-heurísticas implementadas foram testadas com o benchmark da Universidade de Toronto.

Palavras-Chave:

- Pesquisa Local
- COMET
- Resolução de Horários
- Programação com Restrições
- Heurísticas

Abstract

The timetabling problem is a highly combinatory problem given the constraint set involved and their interdependencies, as well as the preferences set and the quality characteristics which are difficult to specify end even harder to quantify.
Given the exponential size of the search space it becomes unfeasible to explore it all. Thus, the local search suits, even more since the neighborhood definition can be done in a quite intuitive way, allowing validation and contrast with manual methods of problems solving.
New programming languages and tools for local search have emerged lately, being a useful help in the local search specification declaration, which have been revealed highly competitive on several kinds of combinatory problems.
In this dissertation, we test this approach in timetabling scheduling. More specifically, constraints type, preferences and quality measures which occur in timetabling specification, are analyzed.
After analyzing the neighborhoods to use in this problem solving, as well as meta-heuristics that have revealed to be more interesting, a prototype was implemented. This prototype was tested with timetable examples produced in our institution and the implemented meta-heuristics were tested with the Toronto University benchmark.

Keywords:

- Local Search
- COMET
- Timetabling
- Constraint Programming
- Heuristics

Índice

Índice de Figuras

Índice de Tabelas

1. Introdução

Nos dias de hoje existem vários problemas de optimização combinatória que vão desde a gestão de cadeias de abastecimento até à calendarização de torneios desportivos [1]. Todos eles com várias variantes em grande escala e com numerosas restrições, que fazem destes problemas, problemas NP difíceis. Estes, variam constantemente a sua estrutura, o que contribui para a sua dificuldade de resolução e para o aparecimento de vários desafios ao nível de pesquisa. Dada a importância destes problemas e o seu impacto nas respectivas instituições, é necessário encontrar soluções de qualidade.
Um destes problemas relaciona-se com a resolução de horários, um assunto amplamente estudado no universo universitário [2]. O problema de resolução de horários tem como principal objectivo distribuir um conjunto de exames ou de disciplinas por um número limitado de posições ou *timeslots*, sendo que existem várias restrições que variam consoante a universidade. Podem existir restrições que estão sujeitas a uma obrigatoriedade e outras preferenciais, sendo possível estabelecer um nível de preferência mediante a restrição específica.

Ao longo das últimas décadas, a programação com restrições tem emergido como uma metodologia fundamental para resolver vários problemas de optimização combinatória, como o referido anteriormente, e novas linguagens de programação especializadas em restrições, como o COMET [3], têm sido desenvolvidos para expressar estas restrições e especificar os procedimentos de pesquisa.

O COMET é uma linguagem de programação orientada para objectos, incluindo abstracções que auxiliam o programador na modelação de problemas de pesquisa local com base em restrições [4], sendo este um dos paradigmas computacionais subjacente ao COMET. Esta ferramenta divide o algoritmo de pesquisa em duas componentes fundamentais: um modelo de alto nível, que descreve a aplicação relativamente às restrições e à função objectivo; e um processo de pesquisa, expressa em termos de modelo num alto nível de abstracção.

A pesquisa local com base em restrições permite a possibilidade de construção de algoritmos autónomos, independentes do método de pesquisa, fomentando a reutilização dos vários módulos em diferentes aplicações e explorando a estrutura do problema, garantindo um desempenho elevado. A utilização de pesquisa local para resolução de problemas de

optimização combinatória tem permitido alcançar soluções admissíveis dentro de um limite de tempo razoável.

Esta dissertação pretende apresentar várias soluções para a produção de horários flexíveis. Estas soluções visam contemplar diferentes métodos heurísticos de pesquisa local utilizando a linguagem COMET. Depois de implementadas as diferentes soluções, procedeu-se aos testes e à comparação das mesmas. No capítulo 2, apresenta-se uma visão geral do estado de arte da resolução de horários e os principais estudos científicos. De seguida, no capítulo 3, é dada uma introdução geral à linguagem COMET com uma maior ênfase sobre as suas características para a programação baseada em restrições para pesquisa local. Seguidamente, no capítulo 4, são introduzidos os conceitos sobre a pesquisa local e as várias meta heurísticas utilizadas. Depois de introduzir os conceitos mais importantes do nosso problema, no capítulo 5, o problema da FCT/UNL é especificado e apresentam-se as características específicas do nosso protótipo. Por fim, no capítulo 6, apresentam-se os resultados obtidos e as conclusões consequentes, no capítulo 7.

2. Resolução de Horários de Exames

O problema de resolução de horários de exames consiste na alocação de um determinado conjunto de exames $E = e_1,e_2,...,e_n$ a um número limitado de posições ou *timeslots* $T = t_1,t_2,...,t_m$ sujeitas a determinadas restrições. As restrições variam consoante a especificidade do problema podendo contradizer-se de acordo com a universidade ou instituição em estudo [5][6].

As restrições dividem-se em dois grupos: obrigatórias ou facultativas. As restrições obrigatórias, não podem ser violadas em nenhuma circunstância, devido à sua elevada importância na resolução do problema. Por exemplo, um determinado aluno ter dois exames no mesmo dia e à mesma hora, um número limitado de alunos por sala, entre outros. Uma solução que respeite estas restrições é uma solução possível para o problema.

As restrições facultativas são restrições que se pretendem satisfazer mas não a todo o custo, podendo ser violadas se necessário. Este tipo de restrições varia consoante o problema específico e a violação das mesmas serve de medida de quantificação da solução encontrada. Normalmente estas restrições estão associadas à dispersão dos exames pelas posições existentes, sendo desejável por vezes dispersar o máximo possível os exames de um mesmo aluno ou colocar todos os exames numa determinada hora do dia.

2.1 Especificações Alternativas

Em 2006, Qu et al [7] fizeram um levantamento dos estudos mais relevantes desta matéria, sumarizando as principais restrições obrigatórias e facultativas que encontraram em toda a literatura estudada. Na Tabela 1 apresentamos um sumário de todas as restrições referidas por estes autores.

<u>Restrições obrigatórias</u>	
1.	Não existência de exames adjacentes com alunos coincidentes.
2.	Garantir que os recursos da posição ou sala são suficientes, i.e. capacidade da sala maior ou igual ao número de alunos inscritos, número de posições/salas suficientes para o número de exames.
<u>Restrições facultativas</u>	
1.	Dispersar os exames o máximo possível ou não em posições ou dias consecutivos.
2.	Grupos de exames que requerem estar na mesma data, posição ou sala.
3.	Exames que têm que ser consecutivos.
4.	Colocar todos os exames, ou os exames com maior número de alunos inscritos, o

	mais cedo possível.
5.	Exames ordenados de acordo com as precedências necessárias.
6.	Limitar o número de alunos e/ou exames numa determinada posição.
7.	Exames que estão em conflito no mesmo dia devem ser colocados o mais próximo possível.
8.	Requisitos temporais, e.g. exames estarem ou não numa posição específica.
9.	Exames podem ser partidos em localizações semelhantes.
10.	Apenas exames com o mesmo número de alunos inscritos podem ser anexados na mesma sala.
11.	Requisitos dos recursos disponíveis, e.g. condições das salas.

Tabela 1 - Tabela síntese de restrições para resolução de horários de exame.

2.2 Benchmarks

O elevado interesse e amplitude do problema da resolução de horários de exame levou à proliferação de vários benchmarks utilizados por diferentes autores de modo a ser possível comparar e qualificar os seus estudos. De seguida iremos descrever e apresentar sucintamente os diversos benchmarks existentes e explicar as diferenças entre eles.

Benchmark da Universidade de Toronto

Em 1996, Carter Laporte e Lee [8] apresentaram 13 problemas reais de alocação de exames, concretamente de três escolas do ensino secundário do Canadá, cinco universidades do mesmo país, uma universidade Americana, uma universidade do Reino Unido e uma universidade da Arábia Saudita.

Foram definidas duas variantes de objectivo:

- Minimizar o número de posições necessárias para o problema (variante 1 na tabela 2);
- Minimizar o custo médio por aluno (variante 2 na tabela 2);

A primeira variante pretende encontrar um horário de exames exequível com o menor número de posições possível. A segunda variante utiliza uma função que calcula o custo do horário gerado. Seja *s* um estudante com 2 exames. O custo da proximidade dos exames é w_s i.e. $w_0 = 16$, $w_1 = 8$, $w_2 = 4$, $w_3 = 2$ e $w_4 = 1$. Através desta função custo pretende-se espaçar ao máximo os exames em conflito dentro de um número limitado de posições.

É necessário ter em conta que a formulação feita pelos autores não contempla dois factores importantes: A capacidade das salas/posições; O facto de dois exames consecutivos para um aluno, que estejam colocados em dias diferentes, é melhor do que dois exames consecutivos no mesmo dia.

Contudo, o factor mais importante deste artigo e deste benchmark foi disponibilizar um primeiro problema standard real à comunidade científica, servindo como ponto de partida para os estudos científicos que se seguiram.

Existe alguma confusão em relação a cinco problemas deste benchmark. Isto porque existe uma segunda versão dos mesmos problemas, causando alguma incerteza quanto aos dados utilizados em alguns estudos que utilizam este benchmark.

A tabela 2 apresenta um quadro síntese dos 13 problemas disponibilizados por Carter e disponíveis num repositório online criado pelo mesmo[1]. De notar que alguns destes problemas têm mais do que uma versão e por isso é também indica a versão do mesmo quando necessário, e.g. car-f-92 I.

Data	Instituto	Períodos	Exames	Alunos	Densidade da Matriz de Conflitos
car-f-92 I	Carleton University,Ottawa	32	543	18419	0.14
car-s-91 I	Carleton University,Ottawa	35	682	16925	0.13
ear-f-83 I	Earl Haig Collegiate Institute, Toronto	24	190	1125	0.27
hec-s-92 I	Ecole des Hautes Etudes Commerciales, Montreal	18	81	2823	0.42
kfu-s-93	King Fahd University, Dharan	20	461	5349	0.06
lse-f-91	London School of Economics	18	381	2726	0.06
rye-s-93	Ryerson University, Toronto	23	486	11483	0.08
sta-f-83 I	St Andrew's Junior High School, Toronto	13	139	611	0.14
tre-s-92	Trent University, Peterborough, Ontario	23	261	4360	0.18
uta-s-92 I	Faculty of Arts and Sciences, University of Toronto	35	622	21266	0.13
ute-s-92	Faculty of Engineering, University of Toronto	10	184	2749	0.08
yor-f-83 I	York Mills Collegiate Institute, Toronto	21	181	941	0.29

Tabela 2 - Quadro Síntese do Benchmark de Toronto

A densidade da matriz de conflitos representa uma estimativa da dificuldade do problema, sendo esta o rácio entre o número de exames com colisões e o número total de exames.

[1] ftp://ftp.mie.utoronto.ca/pub/carter/testprob/

Burke Newall e Weare, em 1996 [9], modificaram este benchmark de forma a poderem testar o mesmo para problemas que consideram a capacidade, com restrições no que concerne à capacidade máxima das posições e exames adjacentes na mesma posição. Em 1998, os mesmos autores [10] voltaram a modificar este benchmark de forma a considerar 3 posições diárias de Segunda a Sexta-feira e uma posição ao Sábado, tendo como objectivo a minimização de exames no mesmo dia ou em dias seguidos para um mesmo aluno.

Existem muitos estudos na literatura que utilizaram o *benchmark* de Toronto para avaliar os resultados obtidos. De seguida apresentamos alguns desses estudos, os quais utilizaremos como comparação com os resultados obtidos nesta investigação.

Caramia, Dell'Olmo e Italiano [11] desenvolveram um método de pesquisa local ambicioso que coloca os exames no menor número possível de períodos utilizando um conjunto de penalizações que procuram melhores soluções sem aumentar o número de períodos. Quando já não existe nenhuma melhoria possível e o horário ainda não representa uma solução possível, o número de períodos é aumentado gradualmente em troca de um elevada penalização. Esta abordagem ainda detém os melhores resultados relatados em várias instâncias do conjunto de dados de Toronto.

Di Gaspero e Schearf [12] realizaram uma investigação utilizando técnicas de pesquisa *tabu* avaliando as vizinhanças com violações de restrições fáceis ou obrigatórias. O tamanho da lista *tabu* é dinâmica e a função de custo é definido de forma adaptativa durante a pesquisa.

Em 2003, Casey e Thompson [13] investigaram procedimentos de pesquisa adaptativos, aleatórios e ambicioso, sendo esta uma técnica relativamente recente na resolução de horários. Neste método, um algoritmo de pesquisa local é iniciado iterativamente depois de alcançar soluções locais óptimas com base nas soluções iniciais geradas por uma abordagem gananciosa. Este método utiliza *backtracking* recorrendo a uma lista *tabu* de modo a evitar ciclos infinitos. Na fase de melhoria da solução foi utilizado o algoritmo de *simulated annealing* com uma temperatura inicial alta e arrefecimento rápido. Esta abordagem aplicada ao conjunto de dados de Toronto devolveu resultados competitivos em alguns dos casos.

Yang e Petrovic [14] utilizaram estudos de casos práticos para escolher heurísticas gráficas para a construção de soluções iniciais para o algoritmo *great deluge* e obtiveram os melhores resultados para diversos casos deste benchmark.

Em 2006, Burke *et al* [15] investigaram variantes de pesquisa de vizinhança variável e obtiveram os melhores resultados em alguns dos problemas deste conjunto de dados. Os resultados foram melhorados, utilizando algoritmos genéticos para seleccionar as vizinhanças.

Pais e Amaral [16] apresentaram uma aplicação da pesquisa *tabu* para o problema de resolução de horários de exame. Estes autores apresentaram um método para gerir automaticamente a memória da lista tabu utilizando um sistema de decisão que se baseia em dois conceitos chave: frequência e inactividade. Estes conceitos estão relacionados, respectivamente, com o número de vezes que uma mudança é introduzida na lista tabu e a

última vez (em número de iterações) que o movimento foi testado e impedido de se concretizar por constar na lista tabu. Este estudo devolve resultados bastante positivos e competitivos em comparação com outros estudos existentes na literatura.

Benchmark da Universidade de Nottingham

Burke, Newall e Weare [9] introduziram também o benchmark da Universidade de Nottingham. Este benchmark contém os dados reais desta Universidade do ano de 1994. Este problema tem um número mínimo de 23 posições dadas as limitações de capacidade das salas. Em 1998 os mesmos autores modificaram também este benchmark de forma a contemplar as restrições no que concerne a exames em dias consecutivos para um mesmo aluno. Este benchmark está disponível num repositório online mantido por Rong Qu[1].

Benchmark da Universidade de Melbourne

O benchmark da Universidade de Melbourne foi introduzido por Merlot et al, em 2003 [17]. Neste benchmark existem dois conjuntos de dados. Este problema define cinco dias de exames com duas posições por dia e com capacidade variável por posição. O objectivo deste problema é minimizar o número de exames consecutivos no mesmo dia ou em dias consecutivos de um mesmo aluno. Este benchmark está disponível num repositório online mantido pela própria universidade[2].

A tabela 3 apresenta uma tabela sumária dos benchmarks descritos com as principais diferenças entre eles.

	Variante	Objectivo
Toronto (Variante 1)	Ignora a capacidade	Minimizar o nº de posições necessárias
Toronto (Variante 2)	Ignora capacidade com custo	Espaçar os exames o máximo possível tendo um nº fixo de posições
Toronto (Burke)	Considera capacidade com custo	Minimizar o número de alunos com 2 exames consecutivos no mesmo dia / em dias consecutivos
Nottingham	Considera capacidade com custo	Minimizar o número de alunos com 2 exames consecutivos no mesmo dia / em dias consecutivos
Melbourne	Considera capacidade com custo	Minimizar o número de alunos com 2 exames consecutivos no mesmo dia / em dias consecutivos

Tabela 3 - Tabela síntese de benchmarks.

[1] http://www.cs.nott.ac.uk/~rxq/data.htm
[2] http://www.or.ms.unimelb.edu.au/timetabling/

Benchmark da FCT/UNL

O benchmark da FCT/UNL é introduzido neste trabalho de investigação. Este contém um conjunto de dados referentes à segunda época de exames, também conhecida como época de recurso, do segundo semestre do ano lectivo 2010-2011.

Foram definidos dois problemas distintos:

- Minimizar o número de alunos com 2 exames consecutivos no mesmo dia / 2 exames não consecutivos no mesmo dia / em dias consecutivos;
- Distribuir os exames pelas diferentes salas disponíveis;

Em relação aos benchmarks utilizados noutros estudos científicos e previamente abordados, comparando com problema específico da FCT/UNL, tratam individualmente os alunos, ou seja, todos eles contêm um ficheiro com as inscrições de cada aluno, ao contrário da FCT/UNL que utiliza apenas uma matriz de conflitos, não fazendo um tratamento individual dos alunos.

3. COMET

COMET [3] é uma linguagem de programação orientada para objectos, especializada na resolução e optimização de problemas com restrições e com recurso a pesquisa local. Um motivo maior para a utilização desta linguagem nesta dissertação, prende-se com facto desta linguagem possuir à partida tipos e classes predefinidos para lidar com problemas de optimização e pesquisa local com restrições. A figura 1 apresenta esquematicamente os tipos de dados definidos no COMET.

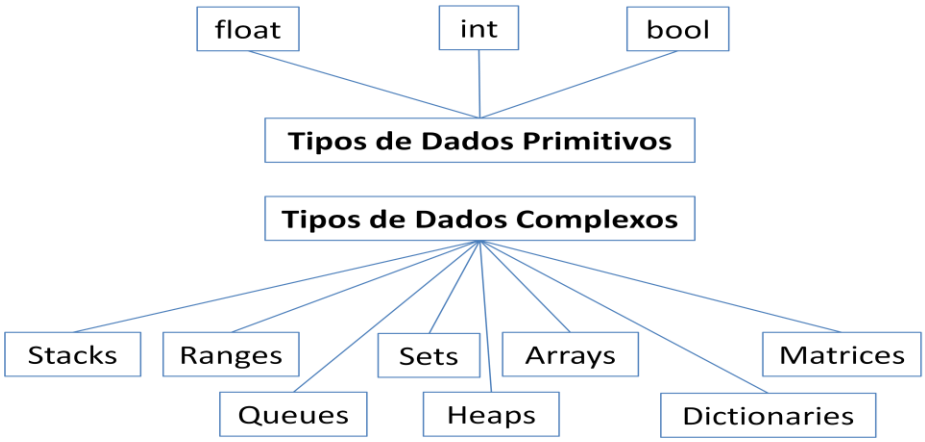

Figura 1 – Tipos de dados na linguagem COMET

3.1 Tipos de Dados Primitivos

Nos tipos de dados primitivos da linguagem COMET incluem-se os números inteiros, os números racionais e os booleanos. Estes tipos de dados, representam-se respectivamente por *int*, *float* e *bool*. A diferença comum entre os tipos de dados primitivos para outros tipos de dados é que os primeiros são passados por valor nos parâmetros de uma função ou método, enquanto os segundos são passados por referência. Excepcionalmente, os tipos de dados primitivos, também podem ser passados por referência se forem declarados como objectos, representados da seguinte forma: *Integer*, *Float*, *Boolean*.

Os números inteiros, representados por *int*, têm as seguintes operações disponíveis: adição (“+”), subtração (“-”), multiplicação (“*”), divisão (“/”), exponencial (“^”) e módulo (“%”).

A linguagem COMET, à semelhança de outras linguagens orientadas para objectos, como por exemplo a linguagem JAVA [18], permite a pré (”++i”) e pós (“i++”) incrementação de

uma variável, assim como a utilização de comparadores (“<”, ”>”, “<=”, “=>”) entre números inteiros.

Os números racionais, representados por *float*, permitem a utilização dos mesmos operadores utilizados para números inteiros, à excepção do operador módulo e dos operadores de incrementação. Este tipo de dados possui outras funções que podem ser úteis na transformação de dados, particularmente, as funções *round*, *floor* e *ceil*. A função *round* arredonda o número para o valor inteiro, maior ou menor, mais próximo. A função *floor* arredonda o número para o valor inteiro, menor, mais próximo, e por fim, a função *ceil*, arredonda o número para o valor inteiro, maior e mais próximo.

Os valores booleanos representam-se por *bool* e podem tomar o valor *true* ou *false*. Para manipular este tipo de dados, os operadores “||” ou “+” funcionam como o operador lógico *or*, e os operadores “&&” ou “*” funcionam como o operador lógico *and*.

3.2 Tipos de Dados Complexos

Nos tipos de dados complexos incluem-se: *Strings*, *Ranges*, *Sets*, *Arrays*, *Matrices*, *Stacks*, *Queues*, *Heaps* e *Dictionaries*. Este tipos de dados, utilizam tipos de dados básicos nas suas estruturas.

O tipos de dados *String* representam cadeias de caracteres, permitindo manipular o conteúdo destas através de diversas operações. É possível concatenar várias *strings* através do comando “+” ou *concat*, subdividir a *string* original em várias *strings* através do comando *split* e através das funções *prefix*, *suffix* ou *substring* podemos obter apenas uma parte da *string*.

Os *Ranges* representam intervalos de valores e estão presentes em grande parte da linguagem COMET, nomeadamente: na declaração de vectores, para definir o intervalo de valores; nos argumentos de *loops*; e na pesquisa local para definir o domínio das variáveis.

O tipo de dados *Set* representa um conjunto de objectos de um determinado tipo de dados. Tomemos como exemplo a seguinte instrução: *set{int} conj = {1,2,3}*. Esta instrução indica a declaração de um conjunto de três objectos do tipo *int*. Estes conjuntos proporcionam várias funcionalidades, como a inserção de novos objectos através do comando *insert*, a possibilidade de apagar um determinado elemento existente com o comando *delete*, utilizar *del* se apenas desejarmos eliminar uma ocorrência desse elemento, ou até mesmo apagar todos os elementos do conjunto através do comando *reset*. Também é possível obter o número de diferentes elementos do conjunto através do comando *getSize*, verificar se um determinado elemento está contido no conjunto através do comando *contains*, ou obter uma cópia do conjunto através do comando *copy*. *Com* este tipo de dados podemos efectuar as operações matemáticas de união e intersecção sobre conjuntos através dos comandos *inter* e *union* respectivamente. Outras das funcionalidades oferecidas pelo *set*, é a capacidade de filtrar ou transformar outros *sets* definidos anteriormente. Tomemos como exemplo a figura 2.

1.	`set{int} aux = {1,2,3,4}`	`{1,2,3,4}`
2.	`set{int} pairs = filter(i in aux)(i%2==0)`	`{2,4}`
3.	`set{int} mult = collect(i in aux)(i*3)`	`{3,6,9,12}`

Figura 2 – Aplicações do tipo de dados *set*.

A figura 2 demonstra como é possível obter os números pares de um *set* previamente definido, utilizando o comando de filtragem *filter*. Com o comando *collect* é possível aplicar uma determinada função a todos os elementos de um *set* existente, como é demostrado na linha 3 da figura 2, que mostra a aplicação do comando *collect* a um *set* denominado *aux*, multiplicando todos os seus elementos pelo valor 3.

Na linguagem COMET, os vectores denominam-se *arrays* e têm associado um *range*, que define o seu intervalo de valores e permite aceder aos vários índices do vector. Através desta associação podemos obter e modificar os diferentes dados pertencentes ao vector não sendo possível alterar a dimensão e os respectivos índices associados ao mesmo.
Existem várias formas de declarar vectores. A figura 3 mostra algumas dessas formas e o respectivo resultado.

1.	`int v1[1..3]`	`v1[0,0,0]`
2.	`int v2[1..3]= [1,2,3]`	`v2[1,2,3]`
3.	`int v3[i in 1..3]= i*2`	`v3[2,4,6]`
4.	`int[] v4`	`(null)`
5.	`v4 = new int[i in 1..3]= i*2`	`anonymous[2,4,6]`

Figura 3 – Aplicações do tipo de dados *array*.

Nos primeiros três exemplos a declaração e a inicialização do vector é feita na mesma instrução, sendo que no primeiro exemplo não são indicados os valores associados aos índices do intervalo de valores especificado, tendo por esse motivo o vector resultante os seus índices com o valor zero. Também é possível declarar um vector sem o inicializar, como exemplificado na linha 4, ou este pode ser inicializado posteriormente como demonstrado na linha 5.

Os vectores permitem efectuar variadas operações entre as quais: *getRange*, que devolve o intervalo de valores do vector; *getSize*, que devolve o número de elementos do vector; *getLow*, que devolve somente o primeiro elemento do vector; *getUp*, que devolve o elemento na última posição do vector; e o método *[i]*, através do qual podemos obter e modificar o elemento na posição *i*.

É possível criar matrizes ou seja vectores multi-dimensionais através da estrutura *array*. Estes vectores são uma extensão dos vectores unidimensionais e utilizam um *range* para cada dimensão representada. Tomemos como exemplo a declaração do seguinte vector multi-dimensional:

```
int matrix[1..3,1..2] = [[3,5],
                         [7,8],
                         [9,10]];
```

Podemos obter o valor de cada posição deste vector indicando os índices da respectiva posição. Por exemplo, a instrução *matrix[1,2]*, devolve o elemento na primeira linha e na segunda coluna, tendo o valor 5.

O tipo de dados *Dictionary* permite definir dicionários que representam um conjunto de elementos chave/valor. Podemos obter um determinado valor contido no dicionário indicando a sua respectiva chave. A chave e o valor podem ser de qualquer tipo de dados, sendo que estes devem ser indicados na criação do dicionário. Podemos obter os valores das chaves através da operação *getKeys*, verificar se uma chave existe através do método *hasKey*, remover uma determinada chave com o comando *removeKey*, obter o número de elementos pertencentes ao dicionário com *getSize* e aceder aos diferentes elementos com o operador *{k}*. A figura 4 descreve um exemplo prático de aplicação dos dicionários.

```
1.  dict{int->string} DicEmpregados()
2.  DicEmpregados{1} = "Pedro"
3.  DicEmpregados{2} = "Miguel"
4.  DicEmpregados{3} = "Joana"
5.  DicEmpregados{4} = "Beatriz"
6.  [<1,Pedro>,<2,Miguel>,<3,Joana>,<4,Beatriz>]
```

Figura 4- Aplicações do tipo de dados *Dictionary*.

As *Stacks* simbolizam pilhas, com uma lógica de *LIFO*, em que o último a entrar é o primeiro a sair. No entanto, na linguagem COMET também é possível remover elementos do fim da pilha através da operação *popBottom*, havendo também a possibilidade de aceder a um determinado elemento directamente através do operador *{i}*.

O tipo de dados *Queues* representam filas onde é possível inserir e remover elementos no princípio e no final. Com os métodos *enqueueFront* e *enqueueBack*, inserimos elementos no princípio e no final da fila, respectivamente. Por outro lado, com os métodos *peekFront* e *dequeBack*, é possível remover elementos do princípio e final da fila. Ao contrário das *Stacks*, nas *Queues* não é possível aceder a um determinado elemento directamente.

As *Heaps* são estruturas de dados semelhantes aos dicionários, na medida em que guardam elementos do tipo chave/valor. Esta estrutura permite aceder apenas à chave, ou ao valor, do menor elemento.

A linguagem COMET suporta as estruturas de controlo de fluxos comuns, nomeadamente, *if*, *for* e *while*. Estas estruturas possuem uma sintaxe em tudo semelhante à presente na linguagem JAVA ou C++ [19]. Para além destas estruturas em COMET, existe também o comando *forall* semelhante ao comando *for* mas que itera os elementos de um *set* ou de um *range*.

Esta linguagem possui também um conjunto de selectores que possibilitam o acesso a elementos de um *set* ou de um *range* através de um critério de selecção pré-estabelecido.

Existem vários tipos de selectores que variam de acordo com o critério de selecção. O comando *select* selecciona um elemento ao acaso. O comando *selectMin* e *selectMax* seleccionam o elemento mínimo e máximo respectivamente, de acordo com uma função de avaliação. O *selectPr* selecciona um elemento de acordo com uma função densidade/probabilidade. O *selectFirst* selecciona o primeiro elemento, por ordem lexicográfica e por fim o *selectCircular*, que selecciona circularmente um elemento do *set* ou *range*, guarda o último elemento seleccionado. Deste modo, ao ser chamado, o selector escolhe o elemento que sucede ao resultado anteriormente seleccionado. Quando este selector é chamado pela primeira vez, ou caso o último elemento do intervalo de valores tenha sido atingido, o selector selecciona o primeiro elemento do intervalo.

3.3 Pesquisa Local em COMET

Inicialmente, em qualquer problema de pesquisa com restrições, na linguagem COMET, é necessário inicializar um objecto do tipo Solver.
Este tipo de objectos guarda internamente todas as variáveis e restrições de um determinado problema e actualiza o seu estado caso exista alguma modificação nas mesmas. Para definirmos um objecto *solver* é necessário declará-lo e, depois de definirmos todo o nosso modelo, e necessário fechá-lo através da instrução *close*. Ao executar esta última instrução, é criado um grafo de dependências, contendo e relacionando todas as variáveis e restrições do problema, sendo necessário de seguida, inicializar as variáveis do mesmo.

Uma das principais características da pesquisa local em COMET é a existência de variáveis incrementais, que são diferenciáveis entre elas. As variáveis incrementais são associadas ao objecto *Solver* do problema, aquando da sua declaração e podem ser do tipo *int*, *float*, *bool* ou *set*. As variáveis podem ser inicializadas recorrendo a um objecto *UniformDistribution* ou *RandomPermutation*, ambos definidos sobre um intervalo de valores, do tipo *range*. Para atribuir um valor à variável através destes objectos, basta usar o comando *get*. O objecto *UniformDistribution*, gera um número aleatório com uma distribuição uniforme, de acordo com o intervalo de valores possíveis para a variável à qual se aplica. Por outro lado o objecto *RandomPermutation*, aplica um valor à variável de acordo com uma permutação aleatória gerada com o intervalo de valores indicado. A permutação não repete os valores atribuídos ao atingir o limite máximo do intervalo de valores indicado, e no caso de se pretender gerar mais valores, será necessário gerar outra permutação utilizando o comando *reset*.

Outra característica importante da pesquisa local em COMET é a existência de invariantes. As invariantes permitem garantir que determinadas condições são mantidas durante toda a execução do problema. Estas são declaradas em associação a variáveis incrementais, sendo que ao atribuir um novo valor à variável incremental associada, esta garante que a condição se mantém. Uma invariante é uma instrução da forma "*v* ← *expression"*, que restringe a variável *v* a manter o valor *expression*. Sempre que as variáveis contidas em *expression* mudam de valor, a variável *v* é actualizada, garantindo a que a

restrição se mantém. As invariantes têm uma natureza declarativa, especificando uma relação sem explicar como se actualiza a mesma. Vejamos o seguinte exemplo:

```
var{int} s(m) ← sum(i in 1..20) i*a[i]
```

Esta invariante garante que a variável *s* é sempre a soma dos elementos do vector *a*, da posição 1, até à posição 20, inclusive. Sempre que o valor na posição *i*, no vector *a*, é modificado, o valor *s* é actualizado automaticamente, fazendo-o corresponder aos valores actuais do vector *a*.

Depois de inicializar todas as variáveis, deverá ser inicializado um objecto de restrições em pesquisa local, denominado por *ConstraintSystem* e associado ao objecto *Solver*. Através deste objecto deverão ser declaradas todas as restrições do problema em questão. Este objecto guarda variadíssimas informações que podem ser acedidas através da interface presente na figura 5.

```
1.  interface Constraint<LS> {
2.  Solver<LS> getLocalSolver()
3.  var{int}[] getVariables()
4.  var{boolean} isTrue()
5.  var{int} violations()
6.  var{int} violations(var{int})
7.  var{int} decrease(var{int})
8.  int getAssignDelta(var{int},int)
9.  int getAssignDelta(var{int},int,var{int},int)
10. int getAssignDelta(var{int}[],int[])
11. int getSwapDelta(var{int},var{int})
12. int getSwapDelta(var{int},var{int},var{int},var{int})
13. void post()
14. }
```

Figura 5 – Interface do sistema de restrições.

Os métodos *getLocalSolver* e *getVariables* devolvem o objecto *solver* e as variáveis incrementais associadas ao objecto *constraint*, respectivamente.

Os métodos seguintes, a partir da linha quatro até a linha sete inclusive, estão relacionados com a satisfação das restrições problema. O método *isTrue* retorna um booleano com valor verdadeiro no caso de todas as restrições do problema serem satisfeitas, ou valor falso no caso contrário. Através dos dois métodos *violations* obtemos o número de violações de todo o problema ou o número de violações associado a uma determinada variável, indicada como argumento. O grupo de métodos que se encontram na linha 8 até à linha 12 inclusive, testam e avaliam o efeito de afectações e trocas de variáveis no número total de violações. Em particular, o método *getAssignDelta(var{int},int)* retorna o aumento do valor de violações provocado pela afectação da variável *var{int}* com o valor *int*. Por outro lado, o método *getSwapDelta(var{int},var{int})* retorna o aumento do valor de violações provocado pela troca de valores entre as variáveis *x* e *y*.

Para um melhor entendimento da pesquisa local em COMET tomemos como problema base o problema das *n*-rainhas. Este problema consiste na colocação de *n*-rainhas num tabuleiro com *n* linhas e *n* colunas, obrigando a que não exista mais do que uma rainha na mesma linha, coluna e diagonal.
Começamos por inicializar um objecto do tipo *solver* e as variáveis fixas que determinam o tamanho do tabuleiro:

```
1. Solver<LS> m()
2. int n = 16
3. range Size = 1..n
```

Figura 6 – Inicialização do objecto *solver*.

De seguida é necessário definir a estrutura de restrições para o nosso problema e um objecto de distribuição uniforme, que é utilizado posteriormente para inicializar as posições das rainhas com valores pseudo-aleatórios.

```
1. ConstraintSystem<LS> S(m)
2. UniformDistribution distr(Size)
3. var{int} queen[Size](m,Size) := distr.get()
```

Figura 7 – Inicialização do sistema de restrições e do tabuleiro *n*-rainhas.

As rainhas são declaradas num vector de números inteiros com *n* elementos. Cada variável, *queen[n]*, representa uma linha do tabuleiro em que a rainha da coluna *n* está posicionada.
Posteriormente, é necessário definir as restrições do nosso problema, sendo que neste caso, as rainhas não podem estar na mesma coluna ou na mesma diagonal.

```
1. S.post(alldifferent(queen))
2. S.post(alldifferent(all(i in Size)(queen[i] + i)))
3. S.post(alldifferent(all(i in Size)(queen[i] - i)))
4. m.close()
```

Figura 8 – Definição das restrições do problema.

Por fim, ao executar o comando *close*, o modelo é fechado.

Depois de concluído o processo de especificação ou caracterização do problema, provém a fase da pesquisa propriamente dita. Para este problema específico, vamos estudar como exemplo um método de resolução que implementa uma heurística simples de maximização/minimização de conflitos.

```
1. while (S.violations() > 0 && it < 50 * n)
2.     selectMax (q in Size)(S.violations(queen[q]))
3.     selectMin (v in Size) (S.getAssignDelta(queen[q],v))
4.     queen[q] := v;
```

Figura 9 – Método de pesquisa para minimização de conflitos.

Como mostra a figura 9, na primeira linha especifica-se o ciclo finito que se pretende efectuar até um determinado limite, que neste caso é de 50 iterações. De seguida, escolhe-se a rainha com o maior número de conflitos na nossa solução actual. Posteriormente, é necessário encontrar a linha que minimiza o número de violações para a rainha escolhida anteriormente e, por fim, atribui-se a linha encontrada à rainha escolhida. Este ciclo é repetido até que seja atingido o limite máximo por nós definido ou caso a solução encontrada não viole qualquer restrição definida.

4. Pesquisa Local

Introduzimos agora uma definição formal de pesquisa local em programação com restrições utilizando definições matemáticas estudadas por Magnus Agren [20].

Tendo um conjunto de variáveis *V*, um domínio *D* e um conjunto de restrições *R*, construímos configurações, que podem ou não ser soluções para o nosso problema.
Uma configuração pode ser definida como uma função $k : V \rightarrow D$. Para uma determinada configuração ser solução de um problema tem que satisfazer o conjunto de restrições *R*.

Os métodos de pesquisa local resolvem os problemas de optimização estudando apenas uma parte espacial do problema, não havendo por isso garantia que se encontre, sempre que exista, uma solução para o problema em estudo.

Para encontrar uma solução melhor do que a solução actual os algoritmos de pesquisa local movimentam-se iterativamente entre possíveis configurações do problema.

Uma movimentação pode ser definida como:

$m : K \rightarrow K$

Dada uma configuração *K*, podemos definir *m(K)* como sendo uma movimentação de *K* ou uma vizinhança de *K*. Os métodos de pesquisa local avaliam várias movimentações possíveis a partir de uma configuração até optar por uma dessas movimentações como sendo a próxima configuração. Deste modo, as configurações que são analisadas constituem a vizinhança da configuração corrente.

Uma função vizinhança pode ser definida como:

$n : K \rightarrow P(K)$

Dada uma configuração *K*, o conjunto de configurações *n(K)* é chamado de vizinhança de *K* e cada elemento deste conjunto é por isso um vizinho de *k*.

Na pesquisa local com recurso a restrições, as penalizações e os conflitos são utilizados para navegar entre possíveis configurações no espaço de pesquisa. Podemos definir penalizações e conflitos entre variáveis como sendo a quantidade de restrições que não são satisfeitas e o quanto as variáveis em particular contribuem para essas violações de restrições.

Uma função de penalização de uma restrição *r* pode ser definida como:

$pen(r) : K \rightarrow N$

Se uma determinada configuração *k* satisfaz o conjunto de restrições *r*, então *pen(r)(k)=0*.
As funções de penalização são usadas de forma a conduzir a pesquisa local em direcção a configurações mais promissoras no espaço de pesquisa de forma a determinar se uma dada configuração é ou não uma solução para o problema. Desta forma, é necessário que uma função de penalização percepcione da melhor forma a semântica da restrição que lhe está

associada. Nesse sentido a definição dada é bastante vaga tal como várias funções de penalização, algumas das quais acabam por não guiar a pesquisa da melhor forma possível.

A figura 10 mostra um grafo como exemplo de um espaço de pesquisa, em que as soluções se representam como nós. Estes estão ligados a outros nós através de arcos, sendo que um conjunto de ligações de um determinado nó representa a sua vizinhança. Cada nó possui um determinado valor, que é o espelho da qualidade do mesmo, fruto da satisfação, ou não, das restrições do problema e de uma ou mais funções objectivo. A pesquisa local pode ser vista em abstracto, como um processo de navegação no grafo, aplicando uma função vizinhança que escolhe um nó vizinho, segundo a qualidade do mesmo, e efectuando a transição do actual estado para o nó escolhido. Este processo de pesquisa termina, idealmente, quando atingimos o nó que representa a solução óptima.

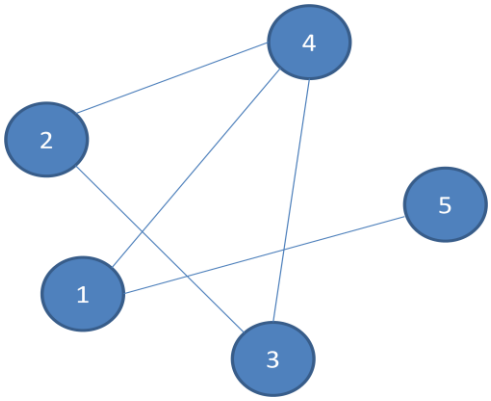

Figura 10 - Grafo de pesquisa local

Existem vários processos de pesquisa local, entre eles, *Hill-Climbing* [21], *Simulated Annealing* [22], *Degraded Ceiling* [23], *Great Deluge* [24] e pesquisa *Tabu* [25] [16]. O que diferencia estes processos são os critérios de aceitação e de rejeição das modificações à solução actual. Entre os algoritmos mais utilizados estão também os algoritmos genéticos [26] mas, dada a sua complexidade e sendo que estes métodos utilizam várias populações, esta dissertação vai apenas abordar métodos com uma população única.

4.1 *Hill-Climbing*

O método mais simples de pesquisa local chama-se *hill-climbing*, que é demonstrado em linguagem COMET na figura 11. Este consiste num algoritmo de pesquisa cujo objectivo é procurar continuamente uma solução de valor superior, aceitando apenas modificações que produzam uma solução pelo menos tão boa como a solução actual. Como o algoritmo não mantém uma árvore de pesquisa, a estrutura de dados do nó da solução actual, só precisa de registar o seu estado e a sua avaliação através do método *S.violations()*.

Uma característica importante é que quando existe mais do que um melhor sucessor para escolher, o algoritmo selecciona o resultado de forma aleatória. Existem três tipos de situações que este tipo de pesquisa pode encontrar:

- Máximo Local: um máximo local é o ponto mais alto apenas em relação à sua vizinhança, existindo pelo menos um ponto maior do que este no espaço de pesquisa. Uma vez num máximo local, o algoritmo irá ficar bloqueado nesse máximo, mesmo que a solução esteja longe de ser satisfatória.

- Planalto: um planalto é uma área do espaço de pesquisa onde a função de avaliação é essencialmente plana. Encontrando-se nesta situação, o algoritmo irá realizar uma pesquisa aleatória.

- Crista: uma crista pode ter uma forte inclinação lateral, deste modo a pesquisa pode atingir a crista com facilidade, mas esta pode inclinar-se muito lentamente em direcção ao ponto mais alto. A menos que existam operadores que se movam directamente ao longo do topo da crista, a pesquisa pode oscilar de um lado para outro, fazendo pouco progresso.

```
1.  function int conflictDirectedSearch(ConstraintSystem<LS> S,
2.     int maxIters){
3.     var{int} [] x = S.getVariables();
4.     int it = 0;
5.     while (S.violations() > 0 && it < maxIters) {
6.        selectMax(i in x.getRange())(S.violations(x[i]))
7.           selectMin(v in x[i].getDomain(),delta =
8.           S.getAssignDelta(examTimeslot[j],v): delta < 0)(delta)
9.              x[i] := v;
10.       it++;
11.    }
12.    return it;
13. }
```

Figura 11 – Pesquisa *hill-climbing* em linguagem COMET.

Em cada um dos casos, o algoritmo atinge um ponto a partir do qual nenhum progresso é concretizado. Se isto acontecer, deve-se reiniciar o algoritmo a partir de um ponto inicial diferente. As reinicializações aleatórias do *hill*-climbing, permitem reiniciar o algoritmo várias vezes a partir de pontos iniciais aleatórios, sendo que o melhor resultado encontrado entre todas as pesquisas é gravado. O critério de paragem pode ser um número fixo de iterações, ou pode ser a não melhoria do resultado em relação a um melhor resultado guardado, para um certo número de iterações.

O sucesso do algoritmo de *hill-climbing* depende muito da forma da superfície do espaço de pesquisa. Caso existam apenas alguns máximos locais e com a ajuda do método de reinicialização aleatória, uma boa solução será encontrada rapidamente [21].

4.2 Pesquisa *Tabu*

A pesquisa *tabu* (PT) tem como estratégia base a exploração do espaço de pesquisa evitando movimentações para configurações que tenham sido testadas num passado recente, de forma a evitar máximos locais. Este algoritmo é baseado em memória e em cada passo da pesquisa, todos os membros da vizinhança são explorados. Entre todos os elementos da actual vizinhança, o membro que devolve o mínimo valor para a função de custo é o escolhido para

o próximo passo. De seguida, a configuração escolhida é adicionada à lista *tabu*, que contém as configurações testadas recentemente. Esta lista é altamente dinâmica, sendo que as configurações *tabu* rapidamente se tornam aceitáveis. Na prática, a lista guarda as configurações *tabu* dentro de um determinado número de iterações a partir do qual podem ser utilizadas novamente.

A figura 12 apresenta o código genérico da pesquisa *tabu* em linguagem COMET. A lista *tabu* é declarada na linha 4 sendo representada por uma estrutura dicionário. Na linha 16 a configuração escolhida é guardada no dicionário, sendo registado juntamente o número de iterações em que se pretende guardar esta configuração como sendo *tabu*.

```
1.  function int constraintDirectedTabu(ConstraintSystem<LS> S,
2.    int tabuLength) {
3.    var{int}[] x = S.getVariables();
4.    dict{var{int} -> int} tabu();
5.    forall (i in x.getRange()) tabu{x[i]} = 0;
6.    int it = 0;
7.    var{int}[] viols = S.getCstrViolations();
8.    while (S.violations() > 0) {
9.      select(j in viols.getRange(): viols[j] > 0){
10.       Constraint<LS> c = S.getConstraint(j);
11.       var{int}[] cx = c.getVariables();
12.       selectMin(i in cx.getRange(), v in cx[i].getDomain():tabu{
13.         cx[i]} <= it)
14.         (S.getAssignDelta(cx[i],v)) {
15.         cx[i] := v;
16.         tabu{cx[i]} = it + tabuLength;
17.       }
18.     }
19.     it++;
20.   }
21.   return it;
22. }
```

Figura 12 – Pesquisa *tabu* em linguagem COMET.

O valor *tabuLength* representa o número de iterações nas quais a configuração é tida como sendo *tabu*. Este valor pode ser fixo, como no exemplo apresentado, ou pode ser dinâmico, podendo por exemplo, variar num intervalo fixado pelo utilizador.

4.3 *Simulated Annealing*

O método *simulated annealing* é um método mais complexo e mais viável, na medida em que pode aceitar modificações que produzam uma solução pior do que a solução actual. O nome *simulated annealing* e os nomes de parâmetro foram escolhidos por uma razão específica. O algoritmo foi desenvolvido a partir de uma analogia explícita com "arrefecimento simulado" - o processo de arrefecimento gradual de um líquido até que este congele. A figura 2 representa a classe genérica deste algoritmo em linguagem COMET e os seus principais métodos. O valor de *delta* corresponde à diferença entre a solução actual e a configuração em avaliação e *temp* corresponde à temperatura. Esta técnica consiste numa pesquisa muito

abrangente no início do processo, com uma probabilidade muito alta de aceitação de configurações com piores soluções, diminuindo gradualmente, quer o espaço de pesquisa quer a aceitação destas movimentações, ao longo de todo o processo de pesquisa. A temperatura é utilizada como parâmetro probabilístico de aceitação de configurações com piores resultados, sendo que vai diminuindo ao longo do tempo.

```
1.  class SimulatedAnnealing {
2.  …
3.    bool localCondition(){
4.      it++;
5.      return it < maxit && ch < maxch;
6.    }
7.    void nextLocal () {
8.      temp = factor * temp;
9.      if (1.0 * ch/it < mp)
10.       fc++;
11.     ch = 0;
12.     it = 0;
13.   }
14.   bool accept(float val) {return expDistr.accept(val/temp);}
15.   bool acceptMove(int delta){
16.     if (delta < 0) {
17.       applyImprove();
18.       return true;
19.     }
20.     else if (delta == 0)
21.       return true;
22.     else if(accept(-delta)){
23.      applyAnnealingAction();
24.      return true;
25.     }
26.     Else
27.       return false;
28.   }
29. …
```

Figura 13 - Pesquisa *simulated annealing* em linguagem COMET.

O *loop* central deste algoritmo é muito semelhante ao de *hill-climbing*. No entanto, em vez de escolher o melhor movimento, é escolhido um movimento aleatório. Se este realmente melhorar a solução, é sempre executado. Caso contrário, o algoritmo faz o movimento com uma determinada probabilidade, que é sempre inferior a 1. A probabilidade decresce exponencialmente com o decréscimo da qualidade do movimento - a quantidade de *delta*, através da qual a avaliação é agravada. A determinação da aceitação desta configuração é feita no método *accept()* com base no seguinte cálculo: *e^-delta/temp*. Em valores mais elevados de *temp*, maus movimentos são mais propícios a serem permitidos. Quando *temp* tende para zero, eles tornam-se cada vez mais improváveis, até que o algoritmo se comporta

de forma semelhante ao *hill-climbing*. O função *nextLocal()* determina o valor de *temp* em função da quantidade de ciclos que já foram concluídos.

Existem vários parâmetros que têm que ser configurados nesta técnica, incluindo a temperatura inicial e final, e o factor de refrescamento. Todos os parâmetros afectam a eficiência deste método, sendo necessário modifica-los consoante o problema, de modo a obter melhores resultados.

4.4 Great Deluge

A pesquisa *great deluge* é muito semelhante à pesquisa *simulated annealing*, diferindo na regra de decisão que determina se um certo passo é aceite ou não. Estas soluções são aceites até ser atingido um valor limite que diminui linearmente ao longo de todo o processo.

O nome deste algoritmo representa uma analogia ao conceito de "grande dilúvio" pela forma como este opera. Tendo como objectivo encontrar o ponto máximo num determinado território, este deixa que a chuva caia no mesmo, inundando-o. Paralelamente, o algoritmo explora o terreno sem nunca ficar abaixo do crescente nível de água. A certa altura, todo o território estará inundado, restando apenas o ponto mais alto do mesmo, que irá corresponder idealmente, à melhor solução.

Neste algoritmo é necessário indicar o nível de água inicial e a velocidade com que chove, ou seja o valor a ser incrementado no nível de água a cada iteração. Uma das grandes vantagens deste algoritmo em comparação com o algoritmo *simulated annealing* é a necessidade de configurar menos parâmetros sendo necessário um conhecimento menos profundo do mesmo para obter bons resultados. A figura 3 mostra a classe genérica da pesquisa *great* deluge e os seus principais métodos em linguagem COMET. O valor de *delta* corresponde à diferença entre a solução actual e a solução em avaliação, *level* representa o nível da água e *factor* a intensidade da chuva. Como podemos observar na linha 6, todas as soluções melhores são aceites e aumentam o nível da água, na linha 12, soluções equivalentes também são aceites mas não alteram o nível da água e na linha 13, caso a qualidade da solução seja pior do que a solução actual, esta pode ser aceite comparando o nível da água em relação a um limite *p* previamente definido.

```
1.  class GreatDeluge {
2.  ..
3.    bool globalCondition(){return ch < maxch;}
4.    void nextLocal () {ch = 0;}
5.    bool acceptMove(int delta){
6.      if (delta < 0) {
7.        level = level + factor;
8.        p = delta + p;
9.        nextLocal();
10.       return true;
11.     }
12.     else if (delta == 0){return true;}
13.     else if (level + delta < p) {
14.       auxx++;
15.       level = level + delta;
16.       nextLocal();
17.       return true;
18.     }
19.     else{
20.       notImprove();
21.       return false;
22.     }
23.   }
24.   void notImprove(){ch++;}
25. …
```

Figura 14 – Pesquisa Great-Deluge em linguagem COMET.

5. O Problema da FCT

Optou-se por fazer à partida, uma divisão clara de dois problemas: a resolução do horário de exames pelas posições disponíveis e a alocação de salas, sendo que o segundo é claramente um problema secundário, que pode ser resolvido muito mais rapidamente do que o primeiro. Depois de encontrar o melhor horário de exames possível é então altura de distribuir os alunos pelas salas disponíveis.

5.1 Resolução de horários de exames

O problema da FCT pretende encontrar o melhor horário de exames possível tendo em conta as seguintes circunstâncias. Existem 42 posições disponíveis, S_1, $S2_,$ …, S_{42}, três por dia, nomeadamente, 9h, 13h e 17h, numa semana de cinco dias, e uma posição ao sábado, às 9h. Para qualquer posição existem N_R salas disponíveis, R_1, R_2, …, R_{NR}. Cada sala tem uma lotação máxima que corresponde ao número máximo de alunos que esta pode suportar. Cada posição tem uma lotação máxima que corresponde à soma da lotação máxima de todas as salas disponíveis para o mesmo. O horário a resolver tem início terça-feira dia 5 de Julho, acaba segunda-feira, dia 18 de Julho e pretende-se colocar 364 exames.

5.1.1 Restrições obrigatórias

- Nenhuma posição tem mais alunos do que a sua capacidade máxima.
- Não existem exames nas posições de Sábado e Domingo à excepção da primeira posição de Sábado.

Para fazer face à restrição da capacidade máxima recorreu-se ao uso de uma invariante que é inicializada com a soma da capacidade máxima de todas as salas disponíveis para cada posição. À medida que são atribuídos exames às posições, a invariante é automaticamente actualizada mantendo sempre a capacidade real existente em cada posição. De forma a evitar a sobrelotação das salas e a garantir lugar para todos os alunos inscritos num determinado exame alocado a uma determinada sala, decidimos inicializar esta invariante apenas com 90% da capacidade total existente para cada posição.

As posições de fim-de-semana nas quais não devem existir exames nunca estão disponíveis para selecção aquando da pesquisa.

5.1.2 Restrições facultativas

- Não existe mais do que um exame para o mesmo aluno na mesma posição.

- Não existe mais do que um exame para o mesmo aluno em posições consecutivas, no mesmo dia.
- Não existe mais do que um exame para o mesmo aluno em posições não consecutivas, no mesmo dia.
- O exame está disponível para aquela posição.

Para definir estas restrições foram criadas duas classes do tipo *User Constraints*, sintetizadas na tabela 4. A classe *dif* verifica a existência de exames na mesma posição ou dia para os mesmos alunos. A classe *sessions* verifica se um determinado exame pode ou não ser colocado numa determinada posição. Esta validação é feita através de um ficheiro de input que indica quais as posições disponíveis para cada exame.

Classe	**Peso fixo**	**Peso Variável**
dif (mesma posição)	333	Nº de alunos
dif (mesmo dia, posição consecutiva)	30	Nº de alunos
dif (mesmo dia, uma posição de intervalo)	15	Nº de alunos
dif (dias consecutivos)	5	Nº de alunos
Sessions	555	-

Tabela 4 - Classes de restrição e pesos atribuídos.

De forma a poder medir a qualidade das soluções foi atribuído um peso a cada uma destas restrições. Caso um aluno tenha dois exames na mesma posição existe um peso fixo de 333 multiplicado pelo peso variável que corresponde ao número de alunos com a mesma sobreposição. Caso o exame não seja na mesma posição mas no mesmo dia, em posições consecutivas, existe um peso fixo de 30 multiplicado pelo número total de alunos contemplados neste caso. Caso o exame não seja na mesma posição mas no mesmo dia, com uma posição de intervalo, existe um peso fixo de 15 multiplicado pelo número total de alunos contemplados neste caso. Por último, caso os exames sejam em dias consecutivos existe um peso fixo de 5 multiplicado pelo número total de alunos contemplados neste caso.

Para exemplificar as penalizações de sobreposições de exames tomemos como exemplo a tabela 5, que apresenta as penalizações entre dois exames cujo número de alunos inscritos em ambos é de 100 alunos.

	Posição				
Exame 1	1	1	3	1	1
Exame 2	1	2	1	4	10
Penalizações	333 * 100	30 * 100	15 * 100	5 * 100	0 * 100

Tabela 5 – Exemplo de penalizações para sobreposições de exames.

5.1.3 Dados de Input

Através dos dados disponibilizados pela própria faculdade temos acesso a um ficheiro com uma matriz que contêm o número de alunos inscritos em cada cadeira e outro ficheiro com as

salas disponíveis e a sua lotação máxima. Foi criado *a posteriori* um ficheiro que contém uma tabela com as posições disponíveis para cada exame utilizado pela classe *sessions*. Este ficheiro tem como objectivo permitir restringir certos exames a uma determinada posição, por exemplo, colocar as cadeiras de matemática obrigatoriamente ao Sábado de manhã. O ficheiro contém o número de linhas igual ao número de exames e em cada linha é indicada a cadeira seguida de um número para cada posição. Este número pode ser 1, se o exame estiver disponível para a posição, ou 0 caso o contrário se verifique. A figura 15 mostra um excerto de uma matriz dos alunos inscritos no semestre ímpar de LEI.

	AED - 8145	AM.1E - 7910	AMIIE - 7996	AM.3c - 5004	ALGA.E - 8001
AED - 8145	308	101	65	1	9
AM.1E - 7910	101	418	3	0	249
AMIIE - 7996	65	3	115	0	11
AM.3c - 5004	1	0	0	309	0
ALGA.E - 8001	9	249	11	0	264

Figura 15 – Excerto de matriz de alunos inscrições em LEI no semestre ímpar.

Através da matriz da figura 15 podemos retirar o número de alunos coincidentes em cada cadeira. Por exemplo, se o exame de AED coincidir com a data e hora do exame de AM1, podemos verificar que existem 101 alunos inscritos nas duas cadeiras, e consequentemente com dois exames sobrepostos.

5.1.4 Vizinhança

Nesta dissertação não se investigaram diferentes estruturas de vizinhança e o seu possível efeito. De modo a assegurar a melhor consistência dos resultados experimentais possíveis, foi utilizada a mesma estrutura de vizinhança e a mais comum nos estudos consultados, em todos os testes com os diferentes algoritmos e dados.

Todas as vizinhanças podem ser obtidas substituindo apenas a posição de um determinado exame por outra posição. Uma das vantagens deste tipo de movimentações é a rápida avalização da mesma, isto porque, a cada iteração os nossos algoritmos calculam apenas a diferença na função custo causada por este movimento. Este tipo de avaliação permite aumentar o número de movimentos produzido num determinado período de processamento e deste modo melhorar a performance dos nossos algoritmos.

Uma estrutura de vizinhança mais complexa, com uma análise mais alargada do espaço de pesquisa, pode produzir melhores resultados no mesmo número de movimentos que a estrutura apresentada. Contudo, estes movimentos podem ser mais morosos computacionalmente, o poderia significar fazer menos ou produzir piores resultados num mesmo período de processamento comparando com a estrutura utilizada.

A utilização de diferentes estruturas de vizinhança deverá ser investigada num trabalho futuro.

5.1.5 Algoritmo Inicial de Pesquisa

De forma a melhorar a eficiência da pesquisa e da solução final alcançada foi desenvolvido um algoritmo que é executado no início da pesquisa e que pretende afastar o máximo possível as cadeiras com o maior número de alunos em conflito. Depois de efectuados vários testes com diferentes dados, procurou-se colocar as cadeiras com mais de 900 alunos em conflito com outras cadeiras, na primeira posição de cada dia com um dia de intervalo, ou seja, terça-feira dia 5 de Julho, quinta-feira dia 7 de Julho, Sábado dia 9 de Julho, Segunda-feira dia 11 de Julho, quarta-feira dia 13 de Julho, sexta-feira dia 15 de Julho, e no último dia, segunda-feira dia 18 de Julho. O algoritmo identifica as cadeiras mais conflituosas e distribui-as pelos dias indicados tendo em conta as cadeiras conflituosas já colocadas anteriormente. Caso a cadeira a colocar, num dia específico, esteja em conflito com uma cadeira conflituosa anteriormente colocada, então o algoritmo procura colocá-la noutro dia. Caso a cadeira a colocar entre em conflito com outras cadeiras em todos os dias possíveis para colocação, então esta cadeira é ignorada e procura-se colocar a cadeira conflituosa seguinte.

5.1.6 Heurísticas

5.1.6.1 *Hill-Climbing*

Para este método de pesquisa foi necessário definir o número de iterações e reinicializações desejado. Depois de vários testes realizados, concluiu-se que seria acertado utilizar 3000 reinicializações com um máximo de 4 iterações em cada pesquisa.

5.1.6.4 Pesquisa *Tabu*

Na pesquisa *tabu*, tal como na pesquisa *hill-climbing*, foi necessário configurar o número de iterações, que foram fixadas em 5000. Foi também necessário configurar o comprimento da lista *tabu*. Depois de efectuados vários testes verificou-se que seria mais acertado utilizar uma lista com comprimento de 5 posições.

5.1.6.2 Simulated Annealing

Para esta pesquisa optou-se por utilizar uma temperatura inicial de 1000 com um factor decrescente de 0,95. Nas condições locais conclui-se que, relativamente ao número máximo de iterações numa determinada temperatura, o melhor seria utilizar duas vezes o número de varáveis do nosso problema, e para o máximo de modificações numa determinada temperatura o melhor seria utilizar igual número ao das variáveis do problema. Outro dos parâmetros definidos foi o valor *mp*, que representa o rácio mínimo modificações/iterações, fixado em 0,000001. Quando o rácio fica abaixo deste valor a variável *freeze counter* é incrementada em uma unidade. Relativamente à condição global, optou-se por fixar o máximo do valor *freeze counter* em 3 unidades.

5.1.6.3 Great Deluge

Nesta pesquisa o nível da água foi iniciado com o valor zero aumentando com um factor de 5 unidades. Em relação à condição global, o número de modificações máximas permitidas foi fixado em 25 vezes o valor de variáveis do nosso problema. O limite *p* que é utilizado para a comparação com o nível de água actual, quando estão a ser testados soluções piores do que a solução actual, foi fixado em 0,001 vezes o valor face à qualidade da solução inicial utilizada.

5.2 Alocação de salas

O problema da alocação de salas consiste em distribuir os exames atribuídos a cada posição pelas diversas salas disponíveis. Para este problema, temos à partida, as posições, S_1, $S2_,$ ..., S_{42}, cada *posição* tem salas disponíveis R_1, R_2, …, R_3 e tem exames atribuídos, E_1, E_2, …, E_n, sendo que cada exame tem um número de alunos inscritos.

Pretende-se distribuir os alunos tendo como única restrição difícil a não atribuição de exames a salas com capacidade inferior ao número total de alunos inscritos no mesmo. Os exames são atribuídos de preferência a salas ainda não ocupadas por outros exames, mesmo que estas ainda tenham capacidade disponível.

A figura Figura 16 - apresenta o ciclo principal em linguagem COMET da distribuição dos exames pelas várias salas. Como se pode ver na linha 1, inicialmente procura-se colocar cada exame numa sala com capacidade igual ao número de alunos inscritos, ou o mais próximo possível deste número. Quando o número de alunos excede a capacidade total de qualquer uma das salas, é seleccionada uma sala com o menor número de exames já colocados, como é possível observar na linha 3. Seguidamente, na linha 5, é seleccionada a sala com a maior capacidade total sendo que o número de exames já colocados na mesma, tem que ser obrigatoriamente igual ao mínimo encontrado anteriormente.

```
1.  selectMin(k in R:roomsAux[k]>=std && roomsExs[k]==0)(roomsAux[k]{…}
2.  while(std>0){
3.    selectMin(r in R)(roomsExs[r])
4.      aux=roomsExs[r];
5.    selectMax(k in R:roomsExs[k]==aux)(roomsAux[k]){
6.      if (std >= roomsAux[k]){
7.        std -= roomsAux[k];
8.        roomsAux[k] = 0;
9.      } else {
10.       roomsAux[k] -= std;
11.       std = 0;
12.     }
13.     roomsExs[k]++;
14.     strg += " Room " + k + "/n";
15.   }
16. }
```

Figura 16 - Alocação de salas em linguagem COMET.

6. Resultados Experimentais

Nesta secção expomos os dados obtidos através dos algoritmos e heurísticas implementadas, e comparamos as diferentes heurísticas. Todos os testes foram executados num Intel Core 2 Duo com 2.1GHz.

6.1 Solução Actual

Com base no problema actual, nos dias e respectivas posições a calendarizar, nos 364 exames a colocar, na matriz de conflitos, nas restrições obrigatórias e facultativas e respectivos pesos, concluiu-se que o horário de exames utilizado pela FCT/UNL contêm 57682 violações. A tabela 6 mostra detalhadamente as violações atribuídas à solução actual, assim como o número total de alunos e exames afectados.

	Exames sobrepostos	Exames consecutivos	Exames mesmo dia	Exames dias consecutivos	Total
Violações	19647	10080	2880	25075	57682
Alunos	59	336	192	5015	5602
Pares de Exames	41	75	59	401	576

Tabela 6 - Violações atribuídas à solução actual.

O total de violações reflecte o número total de penalizações atribuídas a esta solução, tendo em conta os pesos correspondentes. Podemos verificar que, com a solução actual, existem 59 alunos com exames sobrepostos, 336 alunos com exames na mesma posição, 192 alunos com exames no mesmo dia mas com uma posição de intervalo e 5015 alunos com exames em dias seguidos. Podemos também verificar que existem 41 pares de exames com alunos coincidentes sobrepostos, 75 pares de exames com alunos coincidentes consecutivos no mesmo dia, 59 pares de exames com alunos coincidentes no mesmo dia mas com uma posição de intervalo e finalmente 401 pares de exames com alunos coincidentes em dias consecutivos.

6.2 Algoritmo de Melhoria Inicial

Nesta secção pretende-se demonstrar a utilidade e os resultados obtidos através da utilização do algoritmo de pesquisa inicial. A tabela 7 mostra os valores estatísticos obtidos através da utilização deste algoritmo. Estes resultados foram obtidos depois de 100 execuções, das quais

63 produziram uma solução de qualidade superior após a execução deste algoritmo. Como podemos ver através da tabela 7, utilizando este algoritmo, podemos obter uma solução de qualidade superior à gerada aleatoriamente, com um valor médio de melhoria de 96350,55, o que é bastante positivo.

Valor Inicial Médio	Valor Final Médio	Valor Médio de Melhoria
730.886,30	634.535,75	96.350,55

Tabela 7 - Resultados do algoritmo de melhoria inicial.

Tendo em conta o resultado positivo obtido, este algoritmo foi utilizado em todas as execuções efectuadas para as diferentes heurísticas implementadas e todos os resultados seguidamente apresentados partem da solução obtida através do mesmo.

6.3 Pesquisa

Foram realizados vários testes modificando os parâmetros mais relevantes dos diferentes métodos de pesquisa de modo a compreender a sua influência no resultado final. De seguida apresentam-se os vários testes realizados. Todos os resultados obtidos com os diferentes parâmetros foram realizados com recurso a 100 execuções de pesquisa.

6.3.1 *Hill-Climbing*

A tabela 8 mostra os resultados obtidos na pesquisa *hill-climbing* com diferentes valores atribuídos ao parâmetro *it*, o número de iterações. Como podemos observar, o aumento do valor deste parâmetro traz um custo temporal acrescido. Relativamente à qualidade da solução, podemos verificar que o aumento da qualidade da solução só é notório até às 3000 iterações.

Hill Climbing		Tempo (ms)				Qualidade			
		μ_t	min_t	ma_t	σ_t	μ	min	Ma	Σ
it	1000	5240,76	5055	5476	71,77	30229,29	27425	32436	1018,91
	2000	10416,04	10140	11060	161,77	27936,84	25415	30818	1057,47
	3000	15815,66	15132	16458	277,39	27487,57	24558	30783	991,50
	4000	20677,82	20171	21575	301,48	27684,80	25028	30406	995,22
	5000	25869,54	25304	27020	340,49	27457,05	24890	30545	1027,61

Tabela 8 - Tabela comparativa da pesquisa *hill-climbing*.

A figura 17 mostra uma execução de uma pesquisa *hill-climbing*. O gráfico contabiliza as violações da solução no eixo do y, numa escala de logaritmo de base 10 e o número de iterações executadas, fixadas em 2000, no eixo do *x*. Como podemos observar, esta pesquisa devolveu uma solução com cerca de 27000 violações para a melhor solução encontrada, resultado bastante inferior em comparação com a solução actual, 57682.

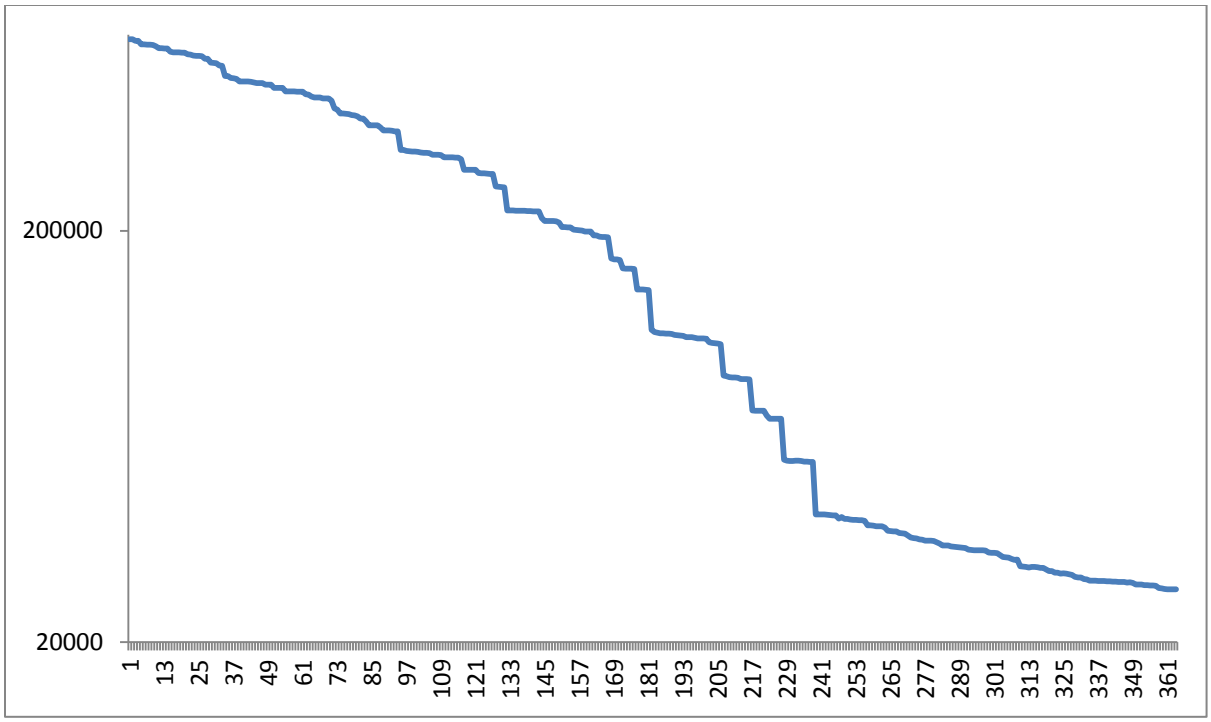

Figura 17 - Gráfico resultante de uma pesquisa *hill-climbing*.

6.3.2 Pesquisa *Tabu*

Na pesquisa *tabu*, foram testadas diferentes configurações inicias, alterando o comprimento da lista *tabu* - *tabuLength* e alterando o número de iterações - *it*. Os resultados obtidos com diferentes valores atribuídos a estes parâmetros foram muito semelhantes no que concerne à qualidade da solução. Contudo, notou-se uma melhoria significativa na qualidade da solução com 5000 ou mais iterações. Quanto ao custo temporal, o aumento do comprimento da lista *tabu* traz um pequeno aumento do mesmo, assim como o aumento das iterações.

Pesquisa *Tabu*		Tempo (ms)				Qualidade			
		μ_t	min_t	ma_t	σ_t	μ	min	ma	σ
tabuLength *it*=4000	3	5650,12	5429	6084	117,14	28826,74	24805	35031	1529,33
	5	5731,30	5491	6490	168,81	28283,91	23991	33688	1769,50
	10	6134,51	5648	6489	144,85	28187,56	25175	32376	1328,76
	30	6100,14	5772	6568	157,95	28258,29	25060	31538	1162,70
it *tabuLength*=5	3000	4527,51	4244	4977	151,21	28653,04	25208	31986	1424,79
	4000	5731,30	5491	6490	168,81	28683,91	23991	33688	1769,50
	5000	7867,88	7551	8190	155,71	27674,97	23920	30290	1342,42
	8000	11509,91	11045	12246	258,94	27094,35	24166	31840	1609,32
	12000	16941,86	16520	17487	174,82	27234,52	24443	30925	1518,41

Tabela 9 - Tabela comparativa da pesquisa *tabu*.

A figura 18 mostra uma execução de uma pesquisa *tabu*. O gráfico contabiliza as violações da solução no eixo do *y*, numa escala de logaritmo de base 10 e o número de iterações executadas, fixadas em 4000, no eixo do *x*. Como podemos observar, esta pesquisa devolveu uma solução com cerca de 26000 violações, resultado bastante inferior em comparação com a solução actual, 57682.

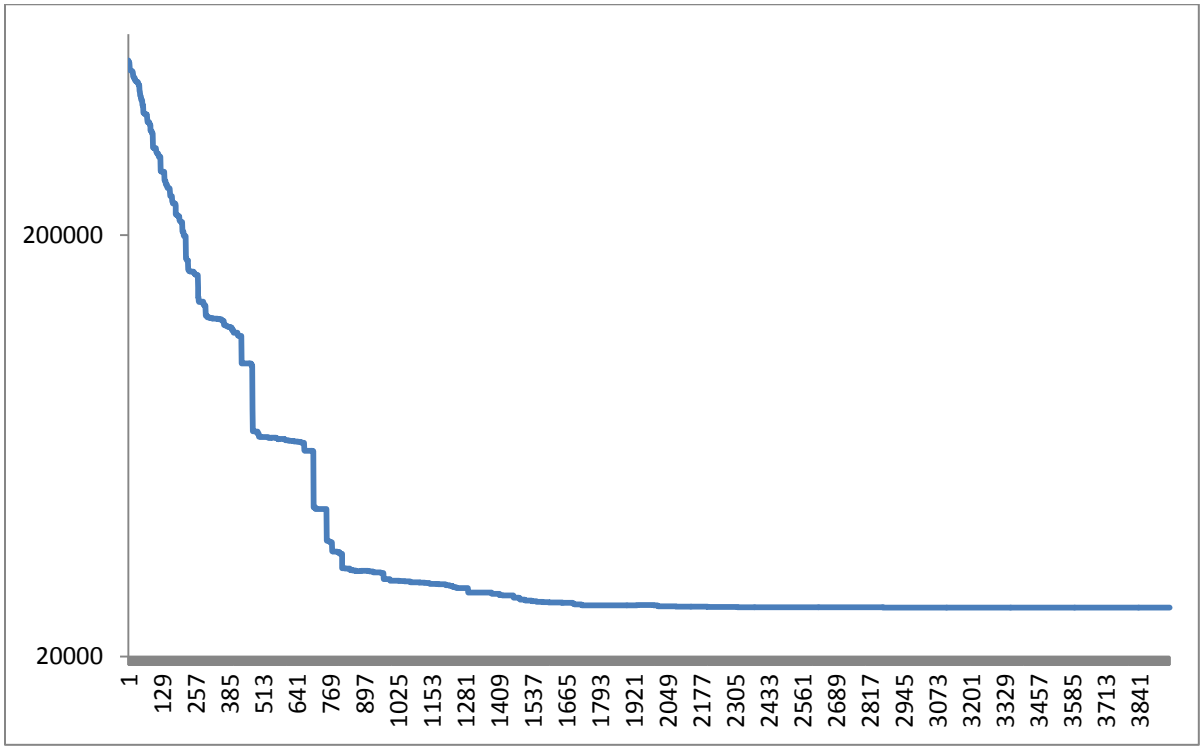

Figura 18 - Gráfico resultante de uma pesquisa *tabu*.

6.3.3 *Simulated Annealing*

A tabela 10 apresenta um quadro síntese dos testes realizados com o método de pesquisa *simulated annealing*. O parâmetro *ch* foi inicializado com os valores indicados, sendo que este factor é multiplicado pelo número de variáveis do problema na sua inicialização. A modificação deste parâmetro não causa um grande impacto na resolução do problema. Com o aumento do valor deste parâmetro, o tempo médio sofre um ligeiro aumento e a solução encontrada melhora ligeiramente.

A modificação do parâmetro *it* tem uma forte influência quer a nível do custo de tempo quer a nível da qualidade da solução. O aumento deste parâmetro acarreta um custo temporal muito forte assim como um aumento significativo na qualidade da solução. Contudo, apesar do aumento temporal ser constante relativamente ao aumento do valor deste parâmetro, o mesmo não acontece com a melhoria da qualidade de solução que vai abrandando o seu aumento. O parâmetro *factor*, tal como o parâmetro *it*, tem uma forte influência no tempo de execução da pesquisa e na qualidade da solução encontrada. Quanto mais próximo de 1 for o valor de *factor*, melhor é a qualidade da solução encontrada. Contudo, este aumento traz um

custo temporal muito significativo. As soluções encontradas com o parâmetro *factor* a 0,99, devolveram as melhores soluções comparando com outras parametrizações da mesma pesquisa e com todas as outras pesquisas estudadas nesta investigação, sendo que o tempo médio de pesquisa quadruplicou comparando com o tempo médio de pesquisa com o *factor* a 0,95.

A modificação do parâmetro *fc* não demonstrou ter um impacto significativo na qualidade da solução encontrada. A qualidade média da solução encontrada manteve-se com os diferentes valores atribuídos a este parâmetro, porém o tempo médio de pesquisa sofreu um ligeiro aumento à medida que o valor do parâmetro foi aumentando.

Por fim, a alteração da temperatura - *temp*, tem um impacto significativo na qualidade da nossa solução, porém, este aumento deixa de ser significativo quando esta é colocada num valor superior a 1000.

Simulated Anneling			Tempo (ms)				Qualidade			
			μ_t	min_t	ma_t	σ_t	μ	min	ma	σ
fc = 3	*ch*	0.5	8137,81	7301	9033	356,80	26011,40	22363	31463	1502,95
it = 2		1	8698,15	7675	9563	381,60	25837,32	23010	30957	1420,65
factor = 0,95		2	8785,68	7972	9781	385,79	25734,40	22915	29521	1338,28
temp = 1000		4	9035,12	8205	10561	481,86	25884,96	23600	28753	1114,64
ch = 1	*it*	1	4352,44	3868	5023	238,32	28393,09	24819	32947	1548,78
fc = 3		2	8698,15	7675	9563	381,60	25837,32	23010	30957	1420,65
factor = 0,95		4	16800,84	15491	18393	584,23	23884,06	21185	27659	1235,64
temp = 1000		8	29797,03	26489	33681	1412,96	23139,07	21223	26930	1101,41
ch = 1	*factor*	0,60	3237,15	1872	4773	453,59	30082,35	26325	35360	1792,32
fc = 3		0,95	8698,15	7675	9563	381,60	25837,32	23010	30957	1420,65
it = 2		0,99	39751,21	36754	43259	1157,23	22203,07	19885	24641	1039,60
temp = 1000		0,999	371207,45	358131	382546	6630,87	19594,80	18015	21213	866,42
ch = 1	*fc*	1	8129,18	7036	9250	374,87	25847,08	22993	28749	1234,44
it = 2		3	8698,15	7675	9563	381,60	25837,32	23010	30957	1420,65
factor = 0,95		9	9512,46	8689	11092	457,09	25659,63	21743	28353	1335,57
temp = 1000		15	10245,16	9236	12308	552,77	25279,33	22450	27249	1110,83
ch = 1	*temp*	100	5623,19	4851	6770	384,37	27920,23	24355	33878	1679,75
fc = 3		500	8270,87	7411	9173	363,66	26297,60	23408	30148	1405,05
it = 2		1000	8698,15	7675	9563	381,60	25837,32	23010	30957	1420,65
factor = 0,95		2000	11731,11	10655	12636	405,13	25395,35	22298	28524	1272,57
		3000	10697,55	9782	11778	473,82	25217,53	22145	28321	1182,82

Tabela 10 - Tabela comparativa da pesquisa *simulated annealing*.

A figura 19 mostra uma execução de uma pesquisa *simulated annealing*. Os gráficos contabilizam as violações da solução no eixo *y*, numa escala de logaritmo de base 10 e o tempo de execução no eixo *x*. É possível verificar que são aceites várias soluções de qualidade inferior ao longo de toda a pesquisa, evitando obter soluções de máximos locais.

Contudo, também se consegue observar que o valor das soluções de qualidade inferior que são aceites ao longo do tempo, se vão aproximando do valor da qualidade actual. Podemos também observar que esta pesquisa encontra uma solução com cerca de 22000 violações, obtendo os valores mais baixos comparando com todas as outras pesquisas implementadas nesta investigação.

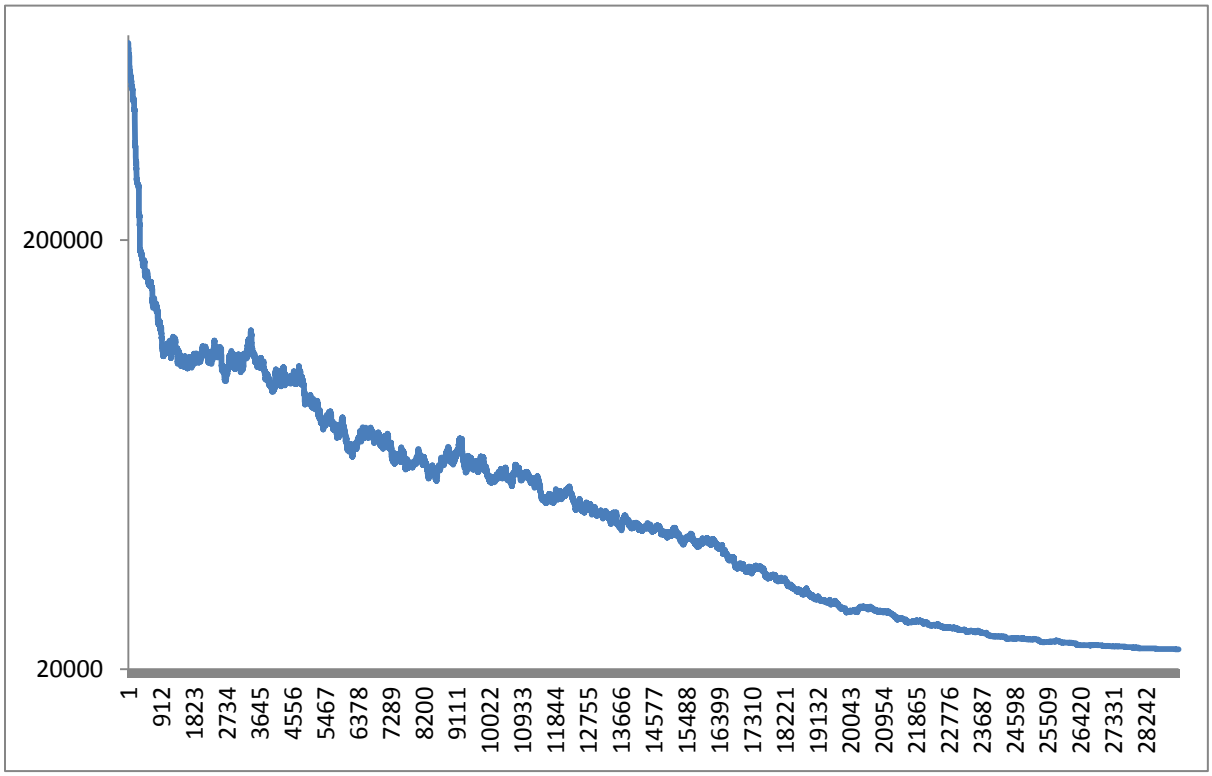

Figura 19 - Gráfico resultante de uma pesquisa *simulated annealing*.

6.3.4 *Great Deluge*

A tabela 11 apresenta um quadro síntese dos testes realizados com o método de pesquisa *great deluge*. Os diferentes valores atribuídos aos diferentes parâmetros não surtiram um efeito significativo na qualidade da solução final encontrada, à excepção do parâmetro *maxch* que obteve resultados de qualidade inferior quando inicializado com um valor inferior a 25. Em termos de custo temporal, a situação anterior manteve-se, ou seja, a modificação do valor do parâmetro *maxch* foi o único que teve um efeito significativo no custo temporal. À medida que o valor deste parâmetro aumenta, o tempo de execução da pesquisa aumenta proporcionalmente.

Great Deluge			Tempo (ms)				Qualidade			
			μ_t	min_t	ma_t	σ_t	μ	min	Ma	Σ
p = 0,001 *factor* = 5	*maxch*	5	4505,76	2964	6162	756,15	30415,07	25925	35511	1897,86
		25	10541,30	6053	16707	2191,83	28151,99	24033	33258	1494,02
		50	14774,36	8393	24976	3539,74	27891,73	23988	33998	1755,27
		100	21978,53	13088	39422	5028,37	28275,34	24895	32801	1478,82
factor = 5 *maxch* = 25	*p*	0,1	11762,18	5694	21356	2496,85	28506,62	25196	32838	1559,93
		0,01	10475,24	6349	16973	1953,05	28267,22	24605	32419	1650,30
		0,001	10541,30	6053	16707	2191,83	28151,99	24033	33258	1494,02
		0,0001	12074,24	5819	18049	2581,54	28713,55	24600	34023	1814,43
p = 0,001 *maxch* = 25	*factor*	2	10978,74	5679	18533	2522,54	28794,42	24641	32726	1594,37
		5	10541,30	6053	16707	2191,83	28151,99	24033	33258	1494,02
		50	11935,20	6786	20373	2862,13	28541,35	25820	33791	1587,70
		200	10503,50	6053	16333	2394,14	28721,74	24376	33918	1818,71

Tabela 11 - Tabela comparativa da pesquisa *great deluge*.

A figura 20 mostra a execução de uma pesquisa *great deluge*. Os gráficos contabilizam as violações da solução no eixo do *y*, numa escala de logaritmo de base 10 e o tempo de execução no eixo do *x*. Apesar de não ser muito visível através da análise deste gráfico, esta pesquisa aceita soluções piores em relação à solução actual encontrada pela pesquisa. O número de soluções piores é muito elevado no início da pesquisa e diminui drasticamente ao longo da pesquisa, deste modo é difícil observar este tipo de aceitações, através da análise gráfica. Podemos também observar que esta pesquisa encontra uma solução com cerca de 25000 violações, resultado bastante inferior em comparação com a solução actual, 57682 violações.

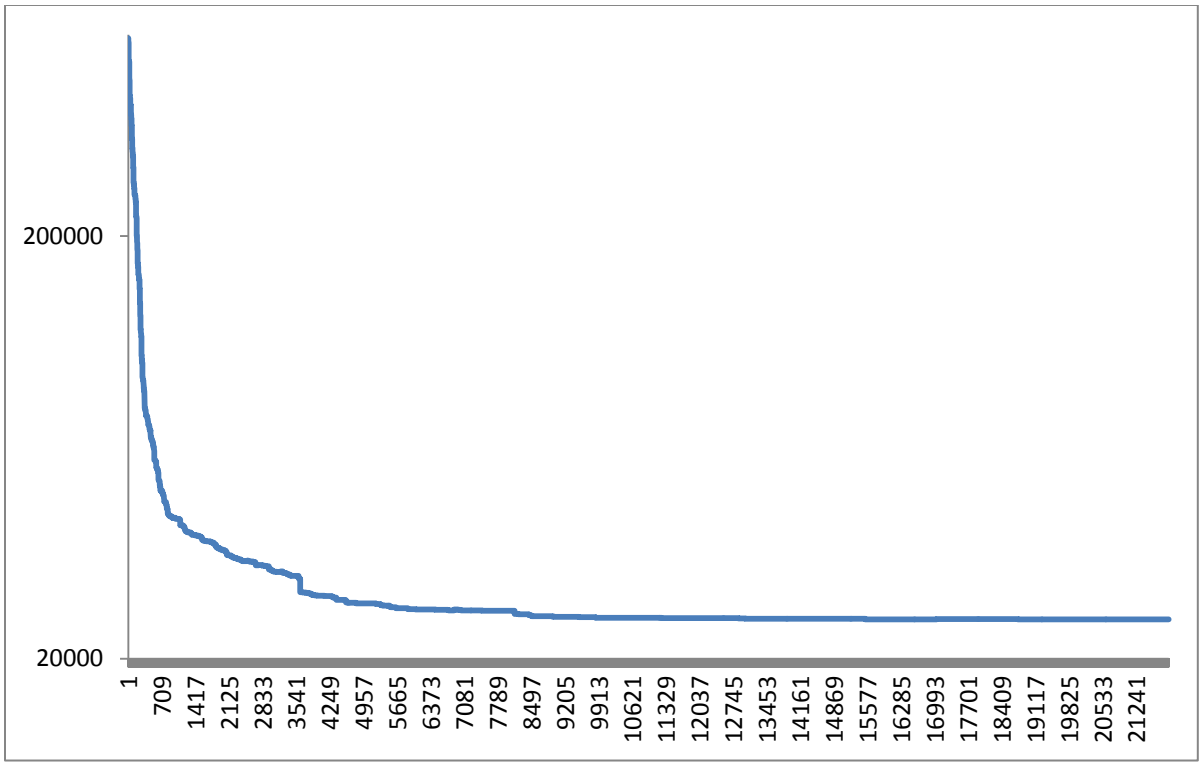

Figura 20 - Gráfico resultante de uma pesquisa *great deluge*.

6.3.5 Comparativo

Nesta secção apresentamos e comparamos os resultados com as diferentes heurísticas implementadas.

A tabela 12 apresenta um quadro síntese estatístico com os melhores resultados encontrados em cada um dos métodos de pesquisa implementados. Podemos verificar que todas as pesquisas devolveram resultados de qualidade superior à solução actual. Contudo, a pesquisa *simulated annealing* foi a que devolveu os melhores resultados. Esta pesquisa mostrou ser melhor em termos de tempo de execução, sendo, de todos os métodos de pesquisa implementados, a que mais rapidamente chegou a uma solução com cerca de 28000 violações. Em termos de qualidade da solução final, foi também a pesquisa *simulated* annealing que devolveu melhores resultados, com cerca de 19500 violações. Finalmente, em termos de relação tempo de execução / qualidade da solução final, encontramos também com este método de pesquisa os melhores resultados, bastando um tempo médio de cerca de 8 segundos para obter resultados médios na ordem das 25700 violações.

	Tempo (ms)				Qualidade			
	μ_t	min_t	ma_t	σ_t	μ	min	ma	σ
HC	15815,7	15132	16458	277,392	27487,6	24558	30783	991,501
TS	6134,51	5648	6489	144,854	28187,6	25175	32376	1328,76
SA	4352,44	3868	5023	238,32	28393,09	24819	32947	1548,78
	8785,68	7972	9781	385,791	25734,4	22915	29521	1338,28
	371207,45	358131	382546	6630,87	19594,8	18015	21213	866,42
GD	14774,4	8393	24976	3539,74	27891,7	23988	33998	1755,27

Tabela 12 - Comparativo de resultados das diferentes heurísticas.

6.4 Melhoria

Nesta secção pretende-se comparar a solução actual, utilizada pela FCT/UNL, com uma das melhores soluções encontradas nesta investigação, tendo em conta o sistema de restrições utilizado. Na tabela 13 apresenta-se um comparativo entre a solução actual e uma das solução encontrada com o método de pesquisa *simulated annealing*, que mostrou devolver as melhores soluções.

		Exames sobrepostos	Exames consecutivos	Exames no mesmo dia	Exames dias consecutivos	Total
Violações	S. actual	19647	10080	2880	25075	57682
	S. em estudo	0	2220	6045	11790	20055
Alunos	S. actual	59	336	192	5015	5602
	S. em estudo	0	74	403	2358	2835
Pares de Exames	S. actual	41	75	59	401	576
	S. em estudo	0	18	99	292	409

Tabela 13 - Comparativo de violações com solução actual e uma solução encontrada.

6.5 Influência da Capacidade das Salas

De modo a testar a influência da capacidade das salas, foram feitos vários testes com o método de pesquisa *simulated annealing*, modificando apenas a capacidade total das salas para cada posição disponível. A tabela 14 apresenta os resultados alcançados.

Capacidade	Tempo (ms)				Qualidade			
	μ_t	min_t	ma_t	σt	μ	min	Ma	σ
um terço	3047,91	1419	4352	471,44	421447,71	296397	651558	61360,32
Metade	8566,21	7847	9969	368,60	38808,58	33885	45403	2127,85
dois terços	9221,37	7956	10796	473,40	29220,03	26331	32203	1384,62
Original	8698,15	7675	9563	381,60	25837,32	23010	30957	1420,65
Dobro	8879,08	7752	9784	422,05	25263,79	22321	28407	1370,05
Triplo	8736,21	7458	9631	401,81	25334,78	22581	28382	1331,56
Décuplo	8553,66	7831	9422	335,70	25201,59	22819	29018	1213,94

Tabela 14 - Comparativo de resultados com diferentes capacidades para as posições.

Podemos verificar através da leitura da tabela 14, que o aumento da capacidade das salas não tem influência no resultado da qualidade da solução encontrada, significando isto que a faculdade não deverá ter qualquer interesse em aumentar a lotação das salas para a realização dos exames. Contudo, também não será do interesse da faculdade diminuir a capacidade das salas, visto que ao diminuir este valor, o número de violações médio da solução encontrada aumenta.

6.6 Influência do Número de Posições

Tendo em conta os melhores resultados encontrados até este ponto, foram feitos testes aumentando o número de dias disponíveis para o horário de exames e consequentemente o número de posições disponíveis. A tabela 15 mostra os resultados obtidos à medida que se foram disponibilizando dias suplementares. Todos os resultados foram obtidos utilizando uma pesquisa *simulated annealing*, sem alterar os seus parâmetros, mas aumentando os dias disponíveis, até um total de 5 dias a mais, comparando com os dias disponíveis actualmente, contabilizando assim 3 semanas com 5 dias úteis e 2 fins-de-semana com uma posição disponível ao sábado. Como podemos observar, através da análise da tabela, a qualidade média da solução encontrada aumenta gradualmente à medida que o número de posições disponíveis vai aumentando.

posições	Tempo (ms)				Qualidade			
	μ_t	min_t	ma_t	σ_t	μ	min	Ma	σ
42	39751,21	36754	43259	1157,23	22203,07	19885	24641	1039,60
45	47464,80	42448	52900	2988,96	18499,30	16763	20635	935,21
48	46545,35	41715	51808	2151,80	13439,32	11950	15210	823,13
51	59536,28	43103	99607	11470,70	10795,88	8990	13550	949,16
54	51570,21	43914	105145	11368,93	8051,62	6915	9258	576,32
57	57256,46	45817	262113	34683,63	6558,36	5570	8088	560,07

Tabela 15 - Comparativo de resultados com diferente número de posições disponíveis.

A tabela 16 compara duas soluções encontradas através da pesquisa *simulated annealing*, fixando os parâmetros da pesquisa e variando as posições disponíveis. A primeira solução

considera as posições disponíveis actualmente, enquanto a segunda considera mais 5 dias disponíveis, o que representa um total de mais 15 posições.

		Exames consecutivos	Exames no mesmo dia	Exames dias consecutivos	Total
Violações	Sol. em estudo	2220	6045	11790	20055
	Sol. com mais 1 semana	240	1470	4665	6375
Alunos	Sol. em estudo	74	403	2358	2835
	Sol. com mais 1 semana	8	98	933	1039
Pares de Exames	Sol. em estudo	18	99	292	409
	Sol. com mais 1 semana	6	49	172	227

Tabela 16 - Comparativo de violações com uma solução com mais posições.

Podemos concluir que a utilização de mais 5 dias de exames, eliminaria grande parte das violações, ficando estas reduzidas a cerca de um quarto das violações existentes em média com um número de posições actual.

6.7 Alocação de Salas

Sendo que este problema é um problema secundário, sem grande impacto no âmbito global do nosso estudo, procurou-se garantir a distribuição correcta dos exames pelas diversas salas disponíveis num curto espaço de tempo.

Na tabela 17 está contabilizado o tempo de execução de 50 execuções do algoritmo de distribuição de exames, sendo que se obteve uma média temporal de execução, de 207,18 milissegundos, um número bastante pequeno para causar impacto na resolução do problema.

μ_t	207,18
min_t	187
ma_t	266
σ_t	11,94

Tabela 17 - Resultados do algoritmo de alocação de salas.

A Figura 21 apresenta um excerto de exemplo de um possível ficheiro de *output* com a indicação das salas alocadas para cada exame, particularmente para a posição 3, referente ao dia 5 de Julho de 2011, às 17 horas.

```
Posição 3
   Exame 2
      Sala 67
   Exame 23
      Sala 50
      Sala 49
   Exame 41
      Sala 6
   Exame 85
      Sala 56
   Exame 172
      Sala 23
   Exame 250
      Sala 1
      Sala 2
   Exame 313
      Sala 17
      Sala 64
   Exame 364
      Sala 7
```

Figura 21 - Excerto de um exemplo do *output* do algoritmo de alocação de salas.

6.7 Comparativo Benchmark de Toronto

Nesta secção pretende-se comparar os resultados obtidos nesta investigação com resultados obtidos noutros estudos anteriormente apresentados, utilizando o conjunto de dados disponibilizado por Carter.

Optamos por utilizar nesta comparação o método de pesquisa mais promissor e que devolveu os melhores resultados para o problema da FCT/UNL a pesquisa *simulated annealing*.

A tabela 18, apresenta um quadro síntese dos melhores dados obtidos em cada um dos estudos, comparando com a nossa pesquisa de *simulated annealing*.

Data Set Version	COMET *Simulated Annealing*	Carter *et al*. [8]	Caramia *et al*. [11]	Di Gaspero & Schaerf [12]	Casey & Thompson [13]	Yang & Petrovic [14]	Burke *et al*. [15]	Pais & Amaral [16]
car91	4.9	7.1	6.6	6.2	5.4	4.5	4.6	5,46
car92	4.69	6.2	6.0	5.2	4.4	3.93	4.0	4,57
ear83	40.5	36.4	29.3	45.7	34.8	33.7	32.8	33,5
hec92	12.19	10.8	9.2	12.4	10.8	10.83	10.0	10,43
rye92	9.26	7.3	6.8	-	-	8.53	-	-
sta83	162.9	161.5	158.2	160.8	134.9	158.35	159.9	157,29
tre92	8.37	9.6	9.4	10.0	8.7	7.92	7.9	8,71
ute92	26.1	25.8	24.4	27.8	25.4	25.39	24.8	24,99
yor83	38.4	41.7	36.2	41.0	37.5	36.35	37.28	37,06

Tabela 18 - Comparativo Benchmark de Toronto

A configuração utilizada na nossa pesquisa *simulated annealing* faz recurso aos mesmos parâmetros anteriormente utilizados para o problema da FCT/UNL, à excepção do parâmetro do factor decrescente, que foi modificado para 0,9999. Este valor permite alcançar soluções mais promissoras aumentando também o tempo de pesquisa.

Como podemos ver na tabela 18, os resultados obtidos pelo nosso método de pesquisa são muito competitivos, aproximando-se muitas vezes do melhor resultado obtido nos outros estudos.

O estudo apresentado por Pais & Amaral [16] e cujos resultados se apresentam na tabela, implementa vários algoritmos. Na tabela 18 apresentamos o melhor resultado obtido pelos vários algoritmos por eles implementados.

7. Conclusões e Trabalho Futuro

A nossa motivação inicial era estudar as capacidades do COMET referentes à pesquisa local, focando ao máximo o nosso problema nas necessidades e características do horário de exames da FCT/UNL.

Baseado nas heurísticas de pesquisa local existentes e utilizadas por outros autores em problemas semelhantes, decidimos implementar quatro heurísticas em COMET: *hill-climbing*, pesquisa *tabu*, *simulated annealing* e *great deluge*.

Para resolver por completo e da melhor forma o problema em estudo, foi também implementado um algoritmo de pesquisa inicial, que pretende diminuir os conflitos existentes inicialmente, alterando apenas a posição das cadeiras com o maior número de alunos em conflito. Por fim, foi também implementado um algoritmo para distribuir os exames de cada posição pelas diferentes salas existentes.

De modo a qualificar os dados obtidos pelas heurísticas implementadas, foi necessário obter e validar o horário de exames da época de recurso do segundo semestre de 2010/2011 da FCT/UNL.

Os resultados obtidos por todas as heurísticas implementadas, foram muito positivos comparando com a solução actualmente utilizada pela FCT/UNL, no que concerne à qualidade da solução em relação às restrições definidas. Em particular, é de realçar o método de pesquisa *simulated annealing* que devolveu as soluções com melhor qualidade.

Depois de implementadas todas as componentes do nosso problema utilizando a ferramenta COMET, é necessário realçar as vantagens claras da utilização desta ferramenta relativamente à modelação do problema. Utilizando o COMET é possível dividir claramente os vários passos do problema em: a definição do problema; a definição de restrições a utilizar; e as diferentes heurísticas de pesquisa.

Todas as heurísticas aqui implementadas poderão ser utilizadas no futuro para estudar outro tipo de problemas sem necessidade de modificar as mesmas, sendo apenas necessário modificar o valor de alguns parâmetros utilizados pelas diferentes heurísticas de forma a obter os melhores resultados.

Em relação aos benchmarks utilizados noutros estudos científicos e aqui abordados, verifica-se que os resultados obtidos nesta investigação são bastante positivos, comparativamente com outros estudos existentes na literatura. Concluímos que a ferramenta COMET consegue devolver resultados competitivos relativamente a outras ferramentas e a linguagens de programação existentes.

Como trabalho futuro seria curioso implementar outros modelos de problemas de horários, com outras especificações e restrições, seguidas por outros autores, recorrendo aos benchmarks aqui apresentados.

Por fim, seria também desafiante reflectir futuramente, a possibilidade de utilização de outro tipo de estruturas de vizinhança e o seu eventual efeito. Nesta dissertação utilizámos a estrutura mais comum, que consiste na substituição directa da posição alocada a um determinado exame. Num trabalho futuro, seria curioso estudar a possibilidade de utilizar uma vizinhança maior, contendo não apenas um exame, mas um grupo de exames e um conjunto de posições possíveis para os mesmos.

A. Apêndice

A.1 Especificação do Problema e Interface Gráfica

```
import cotls;
//VISUALIZER
import qtvisual;
include "dif";
include "sessions";
include "readSessions";
include "readData";
include "readRooms";
include "fit";
include "InitAlg";

ReadData matrix("C:/Program
Files/Dynadec/Comet/compiler/examples/matriz_final.txt");
ReadRooms table("C:/Program
Files/Dynadec/Comet/compiler/examples/salas2009.txt");
ReadSessions ses("C:/Program
Files/Dynadec/Comet/compiler/examples/sessions42.txt");

int nSlots = 42;
int nSlotsPerDay = 3;
int nDiv = nSlotsPerDay;
range Slots = 1..nSlots;
range SlotsV = 1..(nSlots/nSlotsPerDay);
range RoomsV = 1..3;
range Places = 1..nSlots;
range PlacesV = 1..nSlots+2;
range PlacesUr1 = 14..18;
range PlacesUr2 = 35..39;
set{int} PlacesU = {};
forall(p in PlacesUr1)
PlacesU.insert(p);
forall(p in PlacesUr2)
PlacesU.insert(p);
set{int} PlacesS = Places;
set{int} PlacesA = PlacesS \ PlacesU;
int totCap = abs(9*table.getTotCap()/10);//90%
int cap[Places];
string capS[i in PlacesV] = " ";

// ----> VISUALIZER
CometVisualizer visu();
//VisualDisplayTable T = visu.getDisplayTable("Time-Tabling
Problem",RoomsV,SlotsV);
VisualTextTable T(visu.getVisualizer(), "Time-Tabling
Problem",RoomsV,SlotsV);
```

```
visu.addNotebookPage(T);
// <---- VISUALIZER

forall (f in Slots)
   cap[f] = totCap;

forall(p in PlacesUr1)
   cap[p] = 0;
forall(p in PlacesUr2)
   cap[p] = 0;

range Exams = matrix.getExams();
int[,] matCol = matrix.getMatrix();
int[] nStd = matrix.getNStudents();

Solver<LS> m();
UniformDistribution distr(Places);
var{int} examTimeslot[Exams](m, Places);
forall (e in Exams){
   int j = 0;
   do {
   j=distr.get();
   } while (PlacesU.contains(j));
   examTimeslot[e]:=j;
}

Solution solution(m);

ConstraintSystem<LS> S(m);

forall (e in Exams,f in Exams:f>e)
      if(matCol[e,f]>0)
            S.post(dif(examTimeslot[e],examTimeslot[f],matCol[e,f],nDiv));

S.post(sessions(examTimeslot,ses.getMatrix()));

//invariant freeCapacity
var{int}[] cF_;
cF_= new var{int}[i in Places](m) <- cap[i] - sum(j in Exams : i ==
examTimeslot[j]) nStd[j];

m.close();

// ----> VISUALIZER

string auxS1;
string auxS2;
int auxI;
forall (e in Exams){
      auxI = examTimeslot[e];
      auxS1 = capS[auxI];
      capS[auxI] = auxS1 + IntToString(e) + "\n";
}

auxI = 1;
forall (s in SlotsV, r in RoomsV){
      if (capS[auxI]== "")
```

```
            T.setColor(r,s,"white");
      else{
            T.setColor(r,s,"green");
            T.setToolTip(r,s,capS[auxI].rstrip("\n"));
      }
      auxI++;
}

T.setColor(2,5,"grey");
T.setColor(3,5,"grey");
T.setColor(1,6,"grey");
T.setColor(2,6,"grey");
T.setColor(3,6,"grey");
T.setColor(2,12,"grey");
T.setColor(3,12,"grey");
T.setColor(1,13,"grey");
T.setColor(2,13,"grey");
T.setColor(3,13,"grey");

visu.plotViolations(S.violations(),examTimeslot);
visu.pause();

// <---- VISUALIZER

//include "timetablingGreatDeluge";
include "timetablingSimulatedAnnealing";
//include "timetablingTabuSearch";
//include "timetablingHillClimbing";

// --> VISUALIZER
visu.finish();
// <-- VISUALIZER

fit fillRooms(examTimeslot,Slots);
```

A.2 Pesquisa Local

A.2.1 *Hill-Climbing*

```
int runR = 0;
int run = 0;

int violations = S.violations();

int it = 0;
int itR = 0;

int t1 = System.getCPUTime();

while(S.violations() > 0  && run < 4){
   it = 0;
   InitAlg(matCol, Exams, examTimeslot);

   while (S.violations() > 0 && it < 1000) {
        select(j in Exams: S.violations(examTimeslot[j]) > 0)
```

```
            selectMin(v in PlacesA, delta =
S.getAssignDelta(examTimeslot[j],v): nStd[j]<=cF_[v] && delta < 0)(delta)
                  examTimeslot[j] := v;
        it++;

        if (S.violations() < violations){
             violations = S.violations();
             solution.refresh(m);
             runR = run;
             itR = it;
        }
   }

if (S.violations() > 0)
   with delay(m)
   forall (e in Exams){
        int j = 0;
        do {
             j=distr.get();
        } while (PlacesU.contains(j));
        examTimeslot[e]:=j;
   }
run++;

}
int t2 = System.getCPUTime();

solution.restore();

cout << "itR: "<< itR <<  endl;
cout << "cpu time (ms) = " << t2-t1 <<endl;
cout << "violations = " << S.violations() <<  endl;
cout << "runs = " << run << " Best run = "<< runR <<  endl;
```

A.2.2 Pesquisa *Tabu*

```
var{int}[] x = S.getVariables();
dict{var{int} -> int} tabu();
forall (i in x.getRange()) tabu{x[i]} = 0;
int tabuLength = 5;

int violations = S.violations();

int it = 0;

int t1 = System.getCPUTime();

InitAlg(matCol, Exams, examTimeslot);

while (S.violations() > 0 && it < 4000) {
   select(j in Exams: S.violations(examTimeslot[j]) > 0)
        selectMin(v in PlacesA, delta = S.getAssignDelta(examTimeslot[j],v) :
tabu{examTimeslot[j]}<= it && nStd[j]<=cF_[v])(delta){
             examTimeslot[j] := v;
             tabu{examTimeslot[j]} = it + tabuLength;
        }
```

```
   it++;

   if (S.violations() < violations)
        violations = S.violations();
}

int t2 = System.getCPUTime();

cout << "cpu time (ms) = " << t2-t1 <<endl;
cout << "violations = " << violations <<  endl;
```

A.2.3 *Simulated Annealing*

```
include "SimulatedAnnealing";

function var{int} [] saSearch(Constraint<LS> S,var{int} [] x,set{int} Values,
int[] nS, var{int}[] cF) {
   range X = x.getRange();
   int k = X.getSize();
   float prX[i in X] = 1.0/k;
   Solver<LS> m = S.getLocalSolver();
   var{int} sol[X](m);
   SimulatedAnnealing sa(S,k);
   while (sa.globalCondition()){
        while (sa.localCondition()) {

             int kV = Values.getSize();
             range raux = 1..kV;
             float prV[i in raux] = 0.0;
             forall(i in Values) prV[i] = 1.0/kV;
             selectPr(v in X) (prX[v])
             selectPr(c in Values: nS[v]<=cF[c]) (prV[c]){
                  int cdelta = S.getAssignDelta(x[v],c);

                  if (sa.acceptMove(cdelta))
                     x[v] := c;
             }
        }
        sa.nextLocal();
   }
   return sol;
}

int t1 = System.getCPUTime();

InitAlg(matCol, Exams, examTimeslot);
saSearch(S,examTimeslot,PlacesS,nStd,cF_);

int t2 = System.getCPUTime();

cout << "cpu time (ms) = " << t2-t1 <<endl;
cout << "violations = " << S.violations() <<  endl;

class SimulatedAnnealing {

   //current temperature
```

```
float temp;
//rate of change of the temperature
float factor;
//minimum ratio of changes/iterations. Whenever the ration goes
//below this threshold, the freeze counter is increased
float mp;
//n of iterations at a given temperature
int it;
//n of changes at a given temperature
int ch;
//freeze counter:
int fc;
//max n of iterations within a given temperature
int maxit;
//max n of changes within a given temperature
int maxch;
//max n of freezes, since the last solution was found
int maxfc;
ExponentialDistribution distr;
Constraint<LS> S;

SimulatedAnnealing(Constraint<LS> _c, int _n){
     temp = 1000.0;
     factor = 0.95;
     mp = 0.000001;
     it = 0;
     ch = 0;
     fc = 0;
     maxit = 2 * _n;
     maxch = 1 * _n;
     maxfc = 3;
     S = _c;
     distr = new ExponentialDistribution();
}

bool globalCondition() {
     return fc < maxfc;
}

bool localCondition(){
     it++;
     return it < maxit && ch < maxch;
}

void nextLocal () {
     temp = factor * temp;
     if (1.0 * ch/it < mp)
          fc++;
     ch = 0;
     it = 0;
}

bool accept(float val) {return distr.accept(val/temp);}

bool acceptMove(int delta){
     if (delta < 0) {
          applyImprove();
```

```
            return true;
        }
        else if (delta == 0)
            return true;
        else if(accept(-delta)){
            applyAnnealingAction();
            return true;
        }
        else
            return false;
    }

    void applyAnnealingAction(){ch = ch +1;}

    void applyImprove(){
        ch = ch +1;
        if(S.isTrue())
            fc = 0;
    }

}
```

A.2.4 *Great Deluge*

```
include "GreatDeluge";

function Solution gdSearch(Constraint<LS> S,var{int} [] x,set{int} Values,
int[] nS, var{int}[] cF) {
    range X = x.getRange();
    int k = X.getSize();
    float prX[i in X] = 1.0/k;
    Solver<LS> m = S.getLocalSolver();
    Solution solution = new Solution(m);
    GreatDeluge sa(S,k,S.violations());
    int best = System.getMAXINT();
    while (sa.globalCondition()){
            int kV = Values.getSize();
            range raux = 1..kV;
            float prV[i in raux] = 0.0;
            forall(i in Values) prV[i] = 1.0/kV;
            selectPr(v in X) (prX[v])
            selectPr(c in Values: nS[v]<=cF[c]) (prV[c]){
                int cdelta = S.getAssignDelta(x[v],c);
                if (sa.acceptMove(cdelta))
                   x[v] := c;
            }

        if (S.violations() < best){
            best = S.violations();
            solution.refresh(m);
        }
    }
    return solution;
}

int t1 = System.getCPUTime();
```

```
InitAlg(matCol, Exams, examTimeslot);
gdSearch(S,examTimeslot,PlacesS,nStd,cF_);

int t2 = System.getCPUTime();

cout << "cpu time (ms) = " << t2-t1 <<endl;
cout << "violations = " << S.violations() <<  endl;
```

A.3 Restrições

A.3.1 Distância Entre Exames

```
class dif extends UserConstraint < LS > {

      Solver < LS > m;

      var{int}w_;
      dict{var{int}->int} map;
      var{int}[] vars_;

      //posted: indicates if the constraint has been posted
      bool posted;
      var{int} difference_;
      var{int} dayDiff_;
      //violations: total number of constraint violations
      var{int} violations_;
      //isTrue_: indicates whether the constraint is satisfied
      var{bool} isTrue_;

      dif(var{int}a,var{int}b,int w,int nDiv) : UserConstraint <LS>
(a.getLocalSolver()) {
            m = a.getLocalSolver();
            map = new dict{var{int}->int}();
            map{a} = 0;
            map{b} = 1;
            w_ = new var{int}(m);
            w_ : = w;
            vars_ = new var{int} [0. .1] (m);
            vars_[0] = a;
            vars_[1] = b;
            posted = false;
            post();
      }

      void post() {
            if (!posted) {
                  difference_ = new var{int}(m) <- (abs(vars_[0] -
vars_[1]));
                  dayDiff_ = new var {int}(m) < - (abs(abs((vars_[0] - 1) /
3) - abs((vars_[1] - 1) / 3)));
                  violations_ = new var {int}(m) <- (difference_ == 0) * w_ *
333
                              + (difference_ == 1) * (dayDiff_ == 0) * w_ *
30
```

```
                              + (difference_ == 2) * (dayDiff_ == 0) * w_ *
15
                              + (dayDiff_ == 1) * w_ * 5;
                  isTrue_ = new var {bool}(m) <- (violations_ == 0);
                  posted = true;
            }
      }

      var{int}[] getVariables(){
            return vars_;
      }
      var {bool} isTrue() {
            return isTrue_;
      }
      var{int} violations() {
            return violations_;
      }
      var{int} violations(var{int}x) {
            return (violations_);
      }

      int getAssignDelta(var{int}x, int d) {
            int i = map {
                  x};
            int j = 1 - i;
            int auxDiff = abs(vars_[j] - d);
            int auxDayDiff = abs(abs((vars_[j] - 1) / 3) - abs((d - 1) / 3));
            return (auxDiff == 0) * w_ * 333
                  + (auxDiff == 1) * (auxDayDiff == 0) * w_ * 31
                  + (auxDiff == 2) * (auxDayDiff == 0) * w_ * 15
                  + (auxDayDiff == 1) * w_ * 7
                  - (difference_ == 0) * w_ * 333
                  - (difference_ == 1) * (dayDiff_ == 0) * w_ * 31
                  - (difference_ == 2) * (dayDiff_ == 0) * w_ * 15
                  - (dayDiff_ == 1) * w_ * 7;

      }


   }
```

A.3.2 Períodos Específicos Reservados

```
import cotls;

class sessions extends UserConstraint<LS> {

Solver<LS> m;
range R;
var{int}[] a_;
int[,] b_;
int val;
dict{var{int} -> int} map;
bool posted;
var{int}[] valueCount;
var{int} totalViolations;
```

```
var{bool} isConstraintTrue;

sessions(var{int}[] a, int[,] b) : UserConstraint<LS>(a.getLocalSolver()) {
   m = a.getLocalSolver();
   map = new dict{var{int} -> int}();
   a_ = a;
   b_ = b;
   val = 1;
   R = a.getRange();
   posted = false;
   post();
}

void post() {
   if (!posted) {
   forall (i in R)
        map{a_[i]} = i;
   valueCount = new var{int}[i in R](m) <- (b_[i,a_[i]]<val)*555;
   totalViolations = new var{int}(m) <- sum(i in R) valueCount[i];
   isConstraintTrue = new var{bool}(m) <- (totalViolations == 0);
   posted = true;
   }
}

Solver<LS> getLocalSolver() { return m; }
var{int}[] getVariables() { return a_; }
var{bool} isTrue() { return isConstraintTrue; }

var{int} violations() {
   return totalViolations;
}

var{int} violations(var{int} x) {
   int i = map{x};
   return valueCount[i];
}

int getAssignDelta(var{int} x,int d) {
   int i = map{x};
   return ((valueCount[i]==0)*(b_[i,d]<val)*555) -
((valueCount[i]!=0)*(b_[i,d]>val)*555);
   }

}
```

A.4 Algoritmo Inicial

```
class InitAlg {

InitAlg(int[,] mat,range exams,var{int}[] examTimeslot){
   int maxm = 0;
   int count = 0;
   int tot = 0;
   bool auxb = false;
   int aux = 0;
   int auxt = 0;
```

```
int auxr = 0;
int auxx = 0;
int auxconf = 0;
int saux = 1;
range exsr1 = 1..7;
range exsr2 = 1..5;
int auxExams[exsr1,exsr2];

forall (i in exsr1,j in exsr2)
     auxExams[i,j] = 500;
forall (e in exams){
     maxm = 0;
     count = 0;
     tot = 0;
     aux = 0;
     auxconf = 0;
     forall (f in exams){
          aux = mat[e,f];
          count += aux;
          if (aux > 0)
               auxconf++;
          if (aux > maxm)
               maxm = aux;
     }
     tot = count - maxm;
     if (tot > 900){
          auxt++;
          if (auxt > 7){
               forall(i in exsr1){
                  auxb = false;
                  int deb = 0;
                  forall(j in exsr2){
                       if(auxExams[i,j]!=500 && mat[e,auxExams[i,j]]==0){
                            auxb = true;
                            auxr = j;
                            deb++;
                       }
                       else if(auxExams[i,j]!=500){
                            auxb = false;
                            break;
                       }
                  }
                  if(auxb && auxr<exsr2.getUp()){
                       auxx = examTimeslot[auxExams[i,auxr]];
                       examTimeslot[e] := auxx;
                            auxr = auxr+1;
                            auxExams[i,auxr] = e;
                       break;
                  }
               }

          }
          else{
          auxExams[auxt,1] = e;
          examTimeslot[e] := saux;
          }
          if(saux > 37)
```

```
                saux = 1;
            else if(saux == 31)
                saux += 9;
            else
                saux += 6;
        }
    }
}

}
```

A.5 Alocação de Salas

```
include "readData";
include "readRooms";

class fit {

   range R;
   range V;
   range Slots;
   int[] rooms_;
   int[] nStd_;
   dict{int -> string} map;

   fit(var{int}[] exams, range r){
        Slots = r;
        map = new dict{int -> string}();
        ReadRooms table("C:/Program
Files/Dynadec/Comet/compiler/examples/Salas2009.txt");
        ReadData matrix("C:/Program
Files/Dynadec/Comet/compiler/examples/matriz_final.txt");
        rooms_ = table.getPlacesCap();
        nStd_ = matrix.getNStudents();
        R = table.getRRooms();
        V = exams.getRange();
        string[] names = table.getPlacesNames();
        cout << "--------------------------" << endl;
        forall (i in R)
             map{i} = names[i];
        forall(s in Slots){
             int auxi = sum(i in V: exams[i]==s)1 ;
             if(auxi > 0){
             range auxr = 1..auxi;
             int exs[auxr];
             int i = 1;
                  forall(e in V){
                     if (exams[e]==s)
                          exs[i++]=e;
                  }
             cout << "Slot " << s << endl;
             runWorstFit(exs);
             }
        }
        cout << "--------------------------" << endl;
   }
```

```
   void runWorstFit(int[] exams){
        int[] roomsAux;
        roomsAux = new int[i in R]= rooms_[i];
        int roomsExs[i in R] = 0;
        int aux = 0;
        range re = exams.getRange();
        forall(e in re){
             int std = nStd_[exams[e]];
             cout << "   Exam " << exams[e] << endl;

             selectMin(k in R: roomsAux[k]>=std &&
roomsExs[k]==0)(roomsAux[k]/* with highest capacity with less exams  */){
                  roomsExs[k]++;
                  roomsAux[k]-=std;
                  std=0;
                  cout << "      Room " << k << endl;
             }

             while(std>0){
                  selectMin(r in R)(roomsExs[r]/* with less exams  */)
                     aux=roomsExs[r];
                  selectMax(k in R:roomsExs[k]==aux)(roomsAux[k]/* with
highest capacity */){
                            if (std >= roomsAux[k]){
                                 std -= roomsAux[k];
                                 roomsAux[k] = 0;
                            } else {
                                 roomsAux[k] -= std;
                                 std = 0;
                            }
                            roomsExs[k]++;
                            cout << "      Room " << k << endl;
                  }
             }
        }

   }

}
```

A.6 Leitura de Dados de Input

A.6.1 Matriz de Conflitos

```
class ReadData {

string _file;
int[,] _mat;
int _nExams;
range _rExams;
int[] _nStd;

ReadData() {
   _file = "";
}
```

```
ReadData(string file) {
   _file = file;
   readMatrix();
}

void readMatrix(){
   ifstream file(_file);
   int nExams = file.getInt();//n - matriz n*n
   _rExams = 1..nExams;
   int mat[_rExams,_rExams];
   string s1 = file.getLine();
   s1 = file.getLine();
   int nStd[_rExams];

   forall (e in _rExams){
        forall (f in _rExams)
             mat[e,f] = file.getInt();
        file.getLine();
   }

   forall (e in _rExams)
        nStd[e]=mat[e,e];

   _nStd=nStd;
   _mat=mat;
   _nExams=nExams;
}

range getExams(){
   return _rExams;
}

int[,] getMatrix(){
   return _mat;
}

int[] getNStudents(){
   return _nStd;
}

void readSolution(string fl, var{int}[] exs){
   ifstream file(fl);
   string s1 = "";
   range R = exs.getRange();
   forall (i in R){
        int slot = file.getInt();
        exs[i]:=slot;
        s1 = file.getLine();
   }
}

}
```

A.6.2 Salas

```
class ReadRooms {
```

```
string _file;
int _nRooms;
range _rRooms;
string[] _nPlaces;
int[] _cap;
int _totalCap;

ReadRooms() {
   _file = "";
}

ReadRooms(string file) {
   _file = file;
   readTable();
}

void readTable(){
   ifstream file(_file);
   int nRooms = file.getInt();//n - table n*2
   _rRooms = 1..nRooms;
   string s1 = file.getLine();

   string nPlaces[_rRooms];
   int cap[_rRooms];
   int total = 0;
   forall (e in _rRooms){
   s1 = file.getLine();
   string[] line = s1.split(" ");
   nPlaces[e] = line[0];
   cap[e] = line[1].toInt();
   total += cap[e];
   }

   _nPlaces=nPlaces;
   _cap=cap;
   _nRooms=nRooms;
   _totalCap = total;
}

string[] getPlacesNames(){
   return _nPlaces;
}

int[] getPlacesCap(){
   return _cap;
}

int getNRooms(){
   return _nRooms;
}

range getRRooms(){
   return _rRooms;
}

int getTotCap(){
```

```
   return _totalCap;
}

}
```

A.6.3 Períodos

```
class ReadSessions {

string _file;
int[,] _mat;
int _nExams;
int _nSlots;

ReadSessions() {
   _file = "";
}

ReadSessions(string file) {
   _file = file;
   readMatrix();
}

void readMatrix(){
   ifstream file(_file);
   int nExams = file.getInt();
   file.getLine();
   int nSlots = file.getInt();
   file.getLine();
   int mat[1..nExams,1..nSlots];
   forall(ex in 1..nExams){
        file.getInt();
        forall(sl in 1..nSlots)
             mat[ex,sl] = file.getInt();
   }
   _mat = mat;
   _nExams = nExams;
   _nSlots = nSlots;
}

int[,] getMatrix(){
   return _mat;
}

int[] getSessions(int ex){
   int aux[1.._nSlots];
   forall(i in 1.._nSlots)
        aux[i] = _mat[ex,i];
   return aux;
}

}
```

Bibliografia

[1] P. V. Hentenryck and L. Michel, *Constraint-Based Local Search*. The MIT Press, 2005, p. 442.

[2] P. Kostuch, "The University Course Timetabling Problem with a Three-Phase Approach," in *Practice and Theory of Automated Timetabling V*, Springer Berlin / Heidelberg, 2005, pp. 109-125.

[3] D. Decision, "Comet Tutorial," 2010.

[4] P. V. Hentenryck and L. Michel, "Control Abstractions for Local Search," in *Principles and Practice of Constraint Programming*, 2003, pp. 65-80.

[5] E. K. Burke, D. Elliman, P. Ford, and R. Weare, "Examination timetabling in British Universities: A survey," in *Practice and Theory of Automated Timetabling*, Springer Berlin / Heidelberg, 1996, pp. 76-90.

[6] M. Carter, G. Laporte, and J. Chinneck, "A General Examination Scheduling System," *Interfaces*, vol. 24, 1994.

[7] R. Qu, E. Burke, B. Mccollum, L. T. G. Merlot, S. Y. Lee, and C. R. Qu, "A Survey of Search Methodologies and Automated Approaches for Examination Timetabling A Survey of Search Methodologies and Automated Approaches for Examination," *Science Technical Report No . NOTTCS-TR-2006-4*, 2006.

[8] M. Carter, G. Laporte, and S. Y. Lee, "Examination Timetabling: Algorithmic Strategies and Applications," *Operations Research*, vol. 47, pp. 373–383, 1996.

[9] E. K. Burke, J. P. Newall, and R. F. Weare, "A Memetic Algorithm for University Exam Timetabling," *Springer Lecture Notes in Computer Science*, vol. 1153, pp. 1-8, 1996.

[10] E. K. Burke, J. P. Newall, and R. Weare, "Initialization strategies and diversity in evolutionary timetabling." 1998.

[11] M. Caramia, M. DellOlmo, and G. F. Italiano, "New Algorithms for Examination Timetabling," in *WAE '00 Proceedings of the 4th International Workshop on Algorithm Engineering*, 2001.

[12] L. Gaspero and A. Schaerf, “Tabu Search Techniques for Examination Timetabling,” *Practice and Theory of Automated Timetabling III*, vol. 2079, pp. 104-117, 2001.

[13] S. Casey and J. Thompson, “GRASPing the Examination Scheduling Problem,” *Practice and Theory of Automated Timetabling IV*, vol. 2740, pp. 232-244, 2003.

[14] Y. Yang and S. Petrovic, “A Novel Similarity Measure For Heuristic Selection In Examination Timetabling,” *Practice and Theory of Automated Timetabling V*, vol. 3616, pp. 247-269, 2005.

[15] E. Burke, A. Eckersley, B. Mccollum, S. Petrovic, and R. Qu, “Hybrid Variable Neighbourhood Approaches to University Exam Timetabling,” *Computer Science Technical Report No . NOTTCS*, 2006.

[16] T. C. Pais and P. Amaral, “Managing the tabu list length using a fuzzy inference system : an application to exams timetabling,” in *The 7th International Conference for the Practice and Theory of Automated Timetabling*, 2008, pp. 1-6.

[17] L. T. G. Merlot, N. Boland, B. D. Hughes, and P. J. Stuckey, “A Hybrid Algorithm for the Examination Timetabling Problem,” *Springer Lecture Notes in Computer Science*, vol. 2740, pp. 1-24, 2003.

[18] K. Arnold, J. Gosling, and D. Holmes, *Java(TM) Programming Language, The (4th Edition)*. 2005, p. 928.

[19] B. Stroustrup, *The C++ Programming Language: Special Edition*. 2000, p. 1030.

[20] M. Ågren, “Set Constraints for Local Search,” 2007.

[21] S. Russel and P. Norvig, *Artificial Intelligence - A Modern Approach*. Prentice-Hall, 1995, pp. 111-113.

[22] S. Kirkpatrick, “Optimization by simulated annealing: Quantitative studies,” *Journal of Statistical Physics*, vol. 21, no. 5-6, pp. 498-986, Mar. 1984.

[23] E. K. Burke, Y. Bykov, J. Newall, and S. Petrovic, “A time-predefined local search approach to exam timetabling problems,” *IIE Transactions*, vol. 36, no. 6, pp. 509-528, Jun. 2004.

[24] G. Dueck, “New Optimization Techniques - The Great Deluge Algorithm and the Record-to-Record Travel,” *Journal of Computational Physics*, vol. 104, 1993.

[25] F. Glover and M. Laguna, “Tabu search,” *Journal of computational biology*, vol. 16, no. 12, pp. 1689-703, Dec. 2009.

[26] D. E. Goldberg, *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley, 1989.